Synthesizing DSLs for Few-Shot Learning

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

We study the problem of synthesizing domain-specific languages (DSLs) for few-shot learning in symbolic domains. Given a base language and instances of few-shot learning problems, where each instance is split into training and testing samples, the DSL synthesis problem asks for a grammar over the base language which guarantees that small expressions solving training samples also solve corresponding testing samples, where the notion of expression size varies as a parameter. Furthermore, the grammar must meet syntax constraints given as input. We prove that the problem is decidable for a class of languages whose semantics 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. The proofs yield decision procedures based on tree automaton emptiness algorithms.

1 Introduction

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

When dealing with a large class of concepts C, it is typically impossible to identify a target concept $c \in C$ from just a small set of samples S and hence succeed at few-shot learning, as there could be a large number of concepts that are consistent with the sample. In practice, few-shot symbolic learning, such as program synthesis from examples, data-driven learning of invariants for programs, etc., is successful because *researchers* identify a much smaller class of concepts $\mathcal{H}$, called the *hypothesis class*, and then learning happens over $\mathcal{H}$. Note that such an identification is not necessary when presented with a large amount of data, as in mainstream machine learning. The hypothesis class in symbolic learning is defined using a *language*, often referred to as a domain-specific language (DSL), that captures *typical concepts that are useful in the domain*. For few-shot learning, there is often also a *concept ordering* over $\mathcal{H}$, and few-shot learning algorithms find and report, appealing to Occam's razor, the smallest concept in $\mathcal{H}$ with respect to this ordering that is consistent with the samples S. Typical concept orderings are size or depth of expressions.

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

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

2024. ACM XXXX-XXXX/2024/11-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Formulation of DSL Synthesis. In this paper, we are interested in automatically synthesizing DSLs for few-shot learning. DSL synthesis can enable solving learning problems in new domains without human help— DSLs would be first synthesized from typical learning problems in the domain, followed by learning algorithms that solve problems stemming from the domain by restricting their attention to the DSL.

The first contribution of this paper is a *definition* of the problem of DSL synthesis for solving few-shot learning problems. We ask:

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

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

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

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

The above formulation hence asks for a learning bias for the domain D to be encoded in the DSL. Note that a few-shot learning algorithm that picks the smallest consistent expressions in the DSL $\mathcal{H}$ will in fact solve all the few-shot learning problems that the DSL is learned from. In addition to the above constraints, we also consider an additional constraint $\mathcal{G}$ given as input, which serves as a meta-grammar that constrains the class of allowed DSLs by adding additional bias on the syntax and semantics they use. Meta-grammar constraints could include limitations on the number of variables or macros used or restrictions on the semantics of new functions, e.g., disallowing macros that use disjunction.

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

- **(P1)** First, for each instance $p \in I$, $\mathcal{H}$ must be expressive enough to capture a concept c that solves p, in the sense that c is consistent with both the training and testing examples.
- **(P2)** Second, for each instance $p \in I$, consider the smallest concepts c, according to the concept ordering, that are expressible in $\mathcal{H}$ and which satisfy all the training examples in X_p. Then these concepts c must also satisfy the testing examples in Y_p.
- **(P3)** Third, the definition of the hypothesis class $\mathcal{H}$ in terms of the base language must meet the syntactic constraints given by $\mathcal{G}$.

Intuitively, the second condition demands that for any concept expressible in the base language G that solves the training set X_p, but does not solve the testing set Y_p (for some $p \in P$), the DSL $\mathcal{H}$ must either *disallow* expressing this concept or ensure that the concept is only expressible using large expressions (in terms of the concept order) so that there exists a smaller concept expressible in $\mathcal{H}$ that solves the training set X_p and the testing set Y_p.

Decidable DSL Synthesis Problems. The goal of this paper is to identify general classes of *decidable* DSL synthesis problems. We identify four important DSL synthesis problems of increasing difficulty and establish decidability results for them.

All the problems are fairly general in that they are not tied to any particular logics, programming languages, or underlying theories. Rather, our theorems are formulated for languages whose semantics can be evaluated using memory that is *independent* of the size of expressions but can depend arbitrarily on the size of a structure over which expressions are interpreted. This class of languages has been recently shown to be an expressive class that has decidable learning, i.e., the existence of solutions to few-shot learning instances against a fixed DSL is decidable [Krogmeier and Madhusudan 2022, 2023].

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

We consider four problems, in increasing level of difficulty. We are given in all cases a base grammar and a set of few-shot learning instances I, and meta-grammar constraint $\mathcal{G}$. Let us fix a concept ordering.

Adequate DSL synthesis: Is there a DSL $\mathcal{H}$ satisfying constraints $\mathcal{G}$ such that for every instance in I, there is at least one concept expressible in $\mathcal{H}$ that solves the training and testing sets? If so, construct the DSL.

Adequate DSL synthesis with macros: The same question above, except posed for DSLs defined using grammars with macros.

DSL synthesis: Is there a DSL $\mathcal{H}$ satisfying constraints $\mathcal{G}$ such that for every instance in I, there are concepts in $\mathcal{H}$ that satisfy the training set, and the smallest ones, according to the concept ordering, also satisfy the test set. If so, construct the DSL.

DSL synthesis with macros: The same question above, except posed for DSLs defined using grammars with macros.

The problem of adequate DSL synthesis is the first one to solve, and captures the constraints (**P1**) and (**P3**) mentioned above. It asks whether there is *any* DSL that can express concepts that solve the examples in each input learning instance. It is essentially the DSL synthesis problem where the testing sets are empty, and is therefore independent of the concept ordering. DSL synthesis incorporates the constraint (**P2**), and is the problem we have articulated thus far.

It turns out that standard context-free grammars are not powerful enough to capture a class 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, and where use of this macro in the form $f(e_1, e_2)$ results in the uniform substitution of e_i for x_i in e. It is well known that when the terms substituted are arbitrarily large, context-free grammars cannot capture such languages. We hence also study the adequate DSL synthesis and DSL synthesis problems for macro grammars [Fischer 1968].

Decidability Results. Our second contribution is to show that the four problems identified above, adequate DSL synthesis with and without macros and DSL synthesis with and without macros, are *constructively decidable* for a large class of base grammars and semantics.

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

Our constructions are significantly more complex, both conceptually and in terms of time complexity, than these previous constructions. Here we assume the existence of tree automata which can accurately evaluate the semantics of expressions, and we use them as building blocks to construct tree automata for synthesizing DSLs.

The increase in complexity we encounter in DSL synthesis is related to finding *minimal* expressions that witness the solvability of each given few-shot learning instance. In particular, we need to use alternating quantification on trees to capture the fact that there *exists* a solution 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 building on a *dynamic programming algorithm* for evaluating all expressions derivable in a given DSL, in the order of 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 sets. However, we cannot run the algorithm as we do not have a DSL. But, it turns out that 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 translate the dynamic programming algorithm to a tree automaton that simulates it over arbitrarily large DSLs encoded as trees.

Contributions. This paper makes the following contributions:

- A novel formulation of DSL synthesis which asks for a hypothesis class that biases toward few-shot learning in a domain, using few-shot learning instances as input.
- Decidability results for variants of DSL synthesis using grammars as well as macro grammars over a powerful class of base languages. As far as we know, 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 illustrative examples and applications. In Section 3, we review background and introduce some concepts related to the problem formulation. In Section 4, we present aspects of our formulation of DSL synthesis. In Sections 5 and 6, we introduce the adequate 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 DSL synthesis 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 appendix attached as supplementary material.

2 DSL Synthesis: Examples and Applications

In this section we motivate the *DSL synthesis problem* with examples. Computer science is replete with examples of researchers inventing *languages*, with formal syntax and semantics, that are not necessarily highly expressive, but which are adapted to specific domains. For example, temporal logics such as LTL/CTL/CTL* are adapted to distinguishing good and bad behaviors of reactive systems that interact with environments, e.g., LTL was pioneered by Pnueli for this purpose [Pnueli 1977]. Lamport proposed *stutter-free* temporal logics for describing properties of distributed systems [Lamport 1983]. Other examples include SQL for querying databases, standard libraries for programming languages, and specialized DSLs for program synthesis such as those utilized in learning spreadsheet programs [Gulwani 2011].

The DSL synthesis problem we study corresponds to this ubiquitous problem faced by researchers, as in the cases above, when they construct and formalize new systems of concepts for specific domains. The goal is to represent useful datatypes and computations succinctly. We explore a formulation of this problem based on *learning*, where the goal is to synthesize a hypothesis class given instances of few-shot learning problems. A DSL that is adapted for *few-shot* learning should represent common concepts succinctly so that, given only a small number of examples, the more *succinct* expressions that are consistent with these few examples should solve the learning problem.

2.1 Patterns in Sequences

Finding the next number in a sequence is a common question in aptitude/IQ tests. For example,

$$2, 5, 10, 17, ?$$

where a human finds a pattern such as $n^2 + 1$ to predict the next number to be 26.

In many settings, there is a language of *expressions* that candidates are tested on, and the candidate finds the simplest expression that fits the pattern. This implicit assumption is sometimes important, as there can be many patterns that fit the sequence (high degree polynomials, decimal expansions of special expressions, etc.). For instance, $\langle 41, 43, 47, 53, ? \rangle$ has two reasonable answers, one being 59 as the next prime number, as well as 61, also prime, fitting the expression $n^2 + n + 41$.

In such settings, the human can be seen to learn, during practice sessions, expressions that are allowed in the DSL, and searching this DSL for simplest matching expressions when solving new problems. Our DSL synthesis problem matches this view— it asks for synthesizing a DSL using few-shot learning examples, with solutions, as above. Formally, we can model each few-shot learning problem as input-output examples, with a training set and a singleton test set (e.g., $\{(1, 2), (2, 5), (3, 10), (4, 17)\}$ as training set and $\{(5, 26)\}$ as test set for the first example above). Note that we require that the DSL satisfy the property that for each instance, the simplest expressions in the DSL that satisfy the training set also work for the testing set.

2.2 Feature Engineering for Symbolic Concepts

Symbolic concept learning, including techniques for learning decision trees, Boolean concepts, and symbolic regression, often requires a set of features that are functions of lower-level, base features, where the base features are commonly useful for expressing properties relevant to a domain. For instance, base features may be nonlinear inequalities over numeric variables, which are then used in a symbolic concept learning algorithm that considers Boolean combinations over the inequalities. For example, in learning *inductive invariants* and learning *specifications* of programs [Astorga et al. 2021; Garg et al. 2014; Zhu et al. 2018], effective techniques in the literature combine a set of base features using Boolean combinations, where the base features are engineered by researchers.

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 base features, such that a fixed class of symbolic concepts (e.g. Boolean combinations of features) expressed over these n features solves a given set of few-shot learning problems derived 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].

2.3 Unary Temporal Logic from First-order Logic

Suppose our domain involves classification problems on words over a finite alphabet Σ. Our base language might be first-order logic limited to two variables x_1 and x_2, i.e. $\mathsf{FO}(2)$, with the grammar:

$$\varphi \;\rightarrow\; P(x_i) \;\mid\; x_i < x_j \;\mid\; \varphi \wedge \varphi' \;\mid\; \varphi \vee \varphi' \;\mid\; \neg\varphi \;\mid\; \exists x_i.\,\varphi \;\mid\; \forall x_i.\,\varphi$$

Words over Σ are modeled in a standard way as structures with a linear order $<$ over word positions, together with a successor relation *succ*, and letters $p_i \in \Sigma$ modeled by unary predicates P_i which hold at the corresponding word positions.

Perhaps the domain we care about prioritizes concepts familiar from linear temporal logic, e.g. *eventually* and *next-time* operators. These concepts can be expressed more simply by a DSL that prioritizes *guarded quantification*, e.g. a DSL described by the following grammar:

$$\begin{array}{lcl}
V_1 & \rightarrow & \forall x_2.\,(G \rightarrow V_2) \;\mid\; \exists x_2.\,(G \wedge V_2) \;\mid\; A_1 \\
V_2 & \rightarrow & \forall x_1.\,(G \rightarrow V_1) \;\mid\; \exists x_1.\,(G \wedge V_1) \;\mid\; A_2 \\
A_1 & \rightarrow & P_1(x_1) \;\mid\; \cdots \;\mid\; P_k(x_1) \;\mid\; A_1 \wedge A_1 \;\mid\; A_1 \vee A_2 \;\mid\; \neg A_1 \\
A_2 & \rightarrow & P_1(x_2) \;\mid\; \cdots \;\mid\; P_k(x_2) \;\mid\; A_2 \wedge A_2 \;\mid\; A_1 \vee A_2 \;\mid\; \neg A_2 \\
G & \rightarrow & x_1 = x_2 \;\mid\; x_1 < x_2 \;\mid\; x_2 < x_1 \;\mid\; succ(x_1, x_2) \;\mid\; succ(x_2, x_1)
\end{array}$$

This DSL can express the concepts like “eventually there is a position with label a” using a formula $\exists x_2.\,((x_1 < x_2) \vee x_1 = x_2) \wedge A(x_2)$ or the concept “all subsequent positions in the word have label b” using a formula $\forall x_2.\,(x_1 < x_2) \rightarrow B(x_2)$.

Note that a good DSL for learning in a domain *need not be less expressive* than the base language one begins with. Consider a deep result known as Kamp’s theorem [Kamp 1968], which implies that linear temporal logic (with an *until* operator) has the same expressive power over words as unrestricted first-order logic does. Rather than limiting expressive power over words, linear temporal logic instead *rearranges* the space of first-order expressible concepts to more efficiently express things like *a until b* or *eventually c*, etc.

2.4 Stutter-Invariant Temporal Properties

In temporal logic, stutter invariant properties [Lamport 1983] are those which depend only on the observable state variables of a system and *not on the number of time steps* it stays in a given state. For example, the property that *all requests are eventually responded to*, expressed in LTL as

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

is stutter invariant because any trajectory that satisfies φ will still satisfy φ if we repeat (i.e. stutter) some existing letters, e.g., *abbc* is obtained from *abc* by stuttering the second position once. The trajectories $(u \cdot \mathsf{Req} \cdot v \cdot \mathsf{Resp})^{\omega}$, with $u, v \in \Sigma^*$, are exactly those satisfying φ, and this set is closed under stuttering. Further, for trajectories that do not satisfy φ, stuttering does not change this.

If our domain of interest does not distinguish stuttered words, we might discover a fragment of temporal logic which does not distinguish them either. For example, from a base grammar for temporal logic formulas

$$\varphi ::= p \in P \;\mid\; \mathsf{X}\varphi \;\mid\; \mathsf{F}\varphi \;\mid\; \mathsf{G}\varphi \;\mid\; \varphi \mathsf{U} \varphi' \;\mid\; \varphi \wedge \varphi' \;\mid\; \varphi \vee \varphi' \;\mid\; \neg\varphi$$

we might discover a DSL that omits the next time operator $\mathsf{X}\varphi$, which happens to preserve expressive power if we only consider stutter-invariant properties [Peled and Wilke 1997].

2.5 Functionally-Complete Boolean Formulas

In some cases we may want more power than standard grammars to express some interesting scenarios. The following example illustrates the utility of *macros* for DSL synthesis.

Suppose we are given a set of Boolean functions (with each function presented as its truth table encoded as a sequence of bits) and we want to know if there is a small set of Boolean formulas that can be composed together to make formulas for computing each function.

We can formalize a decision problem for this question by asking whether there exists a small set of *macros* $M_1, \dots, M_k$ which can be composed together in different ways to yield formulas that compute each given function. Specifically, we might ask for a *macro grammar* [Fischer 1968] which defines the macros using nonterminal symbols that can take parameters.

Perhaps we are curious about the existence of a *single macro* which is universal for Boolean functions. We might pose the following decision problem:

Given a set of truth tables for Boolean functions $f_1, \dots, f_m$ over n variables, is there one Boolean macro M taking two inputs which can be composed to compute each f_i?

To specify further, we might ask about the existence and synthesis of a *definition* for the two-parameter macro M in the skeleton below, where parameters are indicated by integers, such that the resulting macro grammar expresses some formula computing each given truth table:

$$S \;\rightarrow\; M(S,S) \;\mid\; x_1 \;\mid\; \cdots \;\mid\; x_n \qquad\qquad M(1,2) \;\rightarrow\; ??$$

It is well known that the production $M(1,2) \;\rightarrow\; \neg(1 \wedge 2)$, i.e. the *nand* gate, yields a system that can compute any Boolean function whatsoever, and therefore all those given as input.

2.6 Library Learning

A recently studied problem related to synthesizing DSLs from the program synthesis literature is *library learning* [Bowers et al. 2023; Cao et al. 2023]. Consider an inductive program synthesizer that solves a class of problems $P = \{I_1, \dots, I_n\}$ in the program synthesis from examples paradigm. In a library learning phase, we can try to *refactor* the solutions to these instances I_p in order to learn common concepts/functions, in terms of a library L, that allow us to express the solutions more compactly. DreamCoder utilizes such refactoring in its *dream phase*, and then when synthesizing programs for new instances, it utilizes the learned functions in L to more effectively search for solutions [Ellis et al. 2023]. Recent work on library learning has used anti-unification and e-graphs to solve this problem modulo an equational theory [Cao et al. 2023].

Library learning is similar to the macro grammar learning we study in this paper. 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 in a macro grammar) such that there is *some* solution for each instance that uses this library. And, of course, most importantly, we prove decidability results for a class of DSL synthesis problems while earlier work does not provide any decidability results.

3 Preliminaries

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

3.1 Alphabets, Tree Grammars, and Tree Macro Grammars

Definition 3.1.1 (Ranked alphabet)**.** A ranked alphabet Δ is a set of symbols with arities given by a function arity : $\Delta \rightarrow \mathbb{N}$. We write Δ^i for the subset of Δ that has arity i and we write x^i to indicate a symbol x has arity i. We use T_Δ to denote the smallest set of terms containing the nullary symbols in Δ and closed under forming new terms using symbols of larger arity. For a set of nullary symbols X disjoint from Δ^0 we write $T_\Delta(X)$ to mean $T_{\Delta\cup X}$. We omit Δ from T_Δ and $T_\Delta(X)$ when clear. We also use *tree* and *term* interchangeably.

Definition 3.1.2 (Tree Macro Grammar)**.** A regular tree macro grammar, or simply *macro grammar*, is a tuple $G = (S, N, \Delta, P)$, where: 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 productions drawn from $N \times T_{\Delta\cup N}(\mathbb{N})$. We often write rules (N, t) as $N \longrightarrow t$ and indicate several rules using vertical bars as usual, e.g. $N \longrightarrow t_1 \mid \cdots \mid t_k$.

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.

Definition 3.1.3 (Tree grammar)**.** A tree grammar is a macro grammar $G = (S, N, \Delta, P)$ for which $\mathsf{arity}(X) = 0$ for all $X \in N$.

We consider only *well-formed* macro grammars in the remainder of the paper, i.e. those for which the right-hand side of any production refers only to the parameters for the nonterminal on the left-hand side (if any).

Definition 3.1.4 (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 $t \in T_{\Delta\cup N}(\{1, \ldots, \mathsf{arity}(X)\})$.

What language of trees does a macro grammar define? As in standard tree grammars we can define the language to be the set of ground terms derivable in a finite number of steps by applying production rules. But consider the following example, which illustrates a subtlety related to the order in which rules can be applied in building derivations.

Example 3.1.1. Consider the macro grammar

$$S \longrightarrow F(H), \qquad F(1) \longrightarrow f(1,1), \qquad H \longrightarrow a \mid b,$$

where integers indicate the parameters for a macro symbol. Observe that we could apply productions to "outermost" macro symbols first, as in $S \Longrightarrow F(H) \Longrightarrow f(H, H) \Longrightarrow f(a, b)$, or we could apply them "innermost" first, as in $S \Longrightarrow F(H) \Longrightarrow F(a) \Longrightarrow f(a, a)$, or we could mix the two. These choices affect the expressive power of macro grammars [Fischer 1968].

For simplicity we will only consider outermost semantics for macro grammars, but the technique and results of this paper apply to the other choices as well.

The *outermost order* semantics means that a rule cannot be applied to rewrite a nonterminal M if it appears as a subterm of another nonterminal N. The only rewrites that apply at any given step in a derivation apply to outermost nonterminals in the structure of the term. This can be formalized using *contexts* (e.g. see [Comon et al. 2007, Chapter 2.1]) by adding the requirement that any context used in defining the derivation relation must not contain nonterminal symbols.

Definition 3.1.5 (Language of a Tree Macro Grammar)**.** The *language* $L(G) \subseteq T_\Delta$ of a macro grammar G is defined in a standard way as 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 of the paper, when we say *grammar* or *macro grammar* we mean a *tree* grammar or *tree* macro grammar.

3.2 Encoding Grammars as Trees, Meta-grammars

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

To encode a grammar $G = (S, N, \Delta, P)$ as a tree t we arrange its productions (N_i, α) along the topmost right-going spine of t, with each α hanging to the left, as shown in Figure 1 for $\Delta = \{a^0, b^0, h^1, g^2\}$, $N = \{N_1^0, N_2^0, N_3^1\}$, and $S = N_1$. We use the symbol "root" to indicate the root of the tree and "end" to indicate there are no more productions at the right edge of the tree. We use symbols "lhs_{N_i}" and "rhs_{N_i}" to distinguish between occurrences of nonterminals in the left-hand and right-hand sides of a production. We use positive integers to indicate the parameters for macro symbols. We write $\Gamma(\Delta, N)$ to denote *grammar alphabets* which we use to encode tree grammars over alphabet Δ and nonterminals N.

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

$$\Gamma(\Delta, N) \coloneqq \Delta \sqcup \{\mathsf{root}^1, \mathsf{end}^0\} \sqcup \{\mathsf{lhs}^2_{N_i}, \mathsf{rhs}^{\mathsf{arity}(N_i)}_{N_i} \;:\; N_i \in N\} \sqcup \{1^0, \ldots, k^0\}.$$

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 Appendix A. We note that when decoding a grammar from a tree we choose to make the nonterminal for the topmost production in the tree the starting nonterminal, and when encoding a grammar (S, N, Δ, P) the first production in the list of productions P will be the topmost production. We often will elide the distinction between a *grammar* and its encoding as a *grammar tree.*

Now that we have a way to encode grammars as trees, we can use regular constraints over grammars to apply generally useful biases to DSLs, e.g., we can require that nonterminals use a bounded number of rules.

Definition 3.2.2 (Meta-grammar constraint)**.** We will use meta-grammars as regular constraints that provide a mechanism for constraining the syntax of synthesized DSLs. A *meta-grammar* $\mathcal{G}$ is simply a tree grammar over a grammar alphabet $\Gamma(\Delta, N)$. Note that the meta-grammar in Figure 1 permits all valid grammar trees over the alphabet, but in general it could be more restrictive, e.g. by limiting the number of productions per nonterminal.

3.3 Tree Automata

We make use of *tree automata* to design algorithms for DSL synthesis. For background on tree automata we refer the reader to [Comon et al. 2007], but we highlight some salient aspects here.

We will use *two-way alternating tree automata.* Intuitively, such automata process an input tree by traversing it both *up* and *down* while branching universally in addition to existentially. The transitions are given by Boolean formulae which describe the valid actions the automaton can take to process and accept its input tree. For a symbol h and state set Q, with $q \in Q$, the transition $\delta(q, h)$ is a positive Boolean formula φ over atoms $Q \times \{-1, 0, \ldots, \mathsf{arity}(h)\}$. The automaton makes a transition by picking a satisfying assignment for the atoms and continuing from the nearby nodes and states designated by true atoms. For example, the formula $(q_1, -1) \wedge (q_2, -1) \vee (q_2, 0) \wedge (q_3, 1)$ requires that the automaton *either* continues at the parent node (designated by -1) from both q_1 and q_2 *or* continues at the current node (designated by 0) in q_2 and the first child in q_3. We sometimes use the names **up**, **stay**, **left**, and **right** for the integer directions -1, 0, 1, and 2.

We assume for convenience that both non-deterministic top-down tree automata and two-way alternating tree automata 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 as described above. For non-deterministic automata, the transition formulas do not use stationary moves, moves to the parent, or universal branching, i.e., multiple states required at a single node. Under these conditions, there is a connection between tree automata and tree grammars: a grammar rule $N \longrightarrow f(N_1, \ldots, N_k) \in P$ corresponds to a transition

Meta-grammar $\mathcal{G}$ over $\Gamma(\Delta, N)$:

$$\begin{array}{rcll} S & \longrightarrow & \mathsf{root}(\mathit{Prod}) & \\ \mathit{Prod} & \longrightarrow & \mathsf{lhs}_{N_i}(\mathit{Term}, \mathit{Prod}) & N_i \in N \\ & | & \mathsf{end} & \\ \mathit{Term} & \longrightarrow & g(\mathit{Term}, \mathit{Term}) & \\ & | & h(\mathit{Term}) & \\ & | & N_3(\mathit{Term}) & \\ & | & \mathsf{rhs}_{N_i} & N_i \in N^{=0} \\ & | & a & \\ & | & b & \end{array}$$

Grammar G:

$$\begin{array}{rcl} N_1 & \longrightarrow & h(N_2) \\ N_2 & \longrightarrow & h(N_1) \mid g(a, b) \end{array}$$

Macro grammar G':

$$\begin{array}{rcl} N_1 & \longrightarrow & N_3(N_2) \\ N_2 & \longrightarrow & a \mid b \\ N_3(1) & \longrightarrow & g(1, 1) \end{array}$$

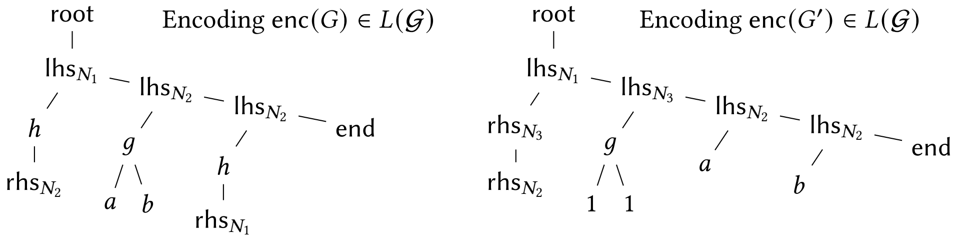


Fig. 1. (Top right) Tree grammars G and G' over $\Delta = \{a^0, b^0, h^1, g^2\}$ and nonterminals $N = \{N_1^0, N_2^0, N_3^1\}$ and (bottom) their encodings as trees $\mathsf{enc}(G)$ and $\mathsf{enc}(G')$ over alphabet $\Gamma(\Delta, N)$. (Top left) A meta-grammar $\mathcal{G}$ over alphabet $\Gamma(\Delta, N)$ with $\mathsf{enc}(G) \in L(\mathcal{G})$.

$\delta(N, f) = (N_1, 1) \wedge \cdots \wedge (N_k, k)$, and given any tree grammar G we can compute in polynomial time a non-deterministic top-down tree automaton $A(G)$ with $L(A(G)) = L(G)$.

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

4 Formulating DSL Synthesis

In this section we formalize and discuss aspects of DSL synthesis which are common across each problem studied in this work, including some terminology and notation that is useful for each.

4.1 Learning Instances, Base Language, Consistency

In this work we introduce and study problems that require synthesizing DSLs given *instances of few-shot learning problems* as input.

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.

The notion of an *example* is flexible, and it is specified by a *base language* using which we can define syntax and semantics for new DSLs.

Definition 4.1.2 (Base language). The *base language* is specified by a tree grammar $G = (S, N, \Delta, P)$, a set $\mathcal{M}$ from which *examples* are drawn, and a predicate consistent $\subseteq L(G) \times \mathcal{M}$ that holds when an expression $e \in G$ is consistent with an example $M \in \mathcal{M}$, written $\mathsf{consistent}(e, M)$. This predicate abstracts away details of specific languages while keeping information relevant for DSL synthesis.

Example 4.1.1 (Propositional Logic)**.** If our base language consists of a grammar for propositional logic formulas φ over variables X, our examples could be variable assignments $\alpha : X \to \{0, 1\}$, each labeled either positive or negative. Then consistency could be defined as

$$\mathsf{consistent}(\varphi, \alpha^+) \Leftrightarrow \alpha \models \varphi \qquad \text{and} \qquad \mathsf{consistent}(\varphi, \alpha^-) \Leftrightarrow \alpha \not\models \varphi.$$

Example 4.1.2 (Regular Expressions)**.** For a base language of regular expressions r over $\Sigma = \{a, b, c\}$, our examples can be labeled words w^+, w^- with $w \in \Sigma^*$. Here, consistency could be defined as

$$\mathsf{consistent}(r, w^+) \Leftrightarrow w \in L(r) \qquad \text{and} \qquad \mathsf{consistent}(r, w^-) \Leftrightarrow w \notin L(r).$$

Example 4.1.3 (Linear Integer Arithmetic)**.** For a base language of linear arithmetic expressions in two variables x, y interpreted over the integers, the examples might be input-output pairs, i.e., $((a, b), o)$, for $a, b, o \in \mathbb{Z}$. Here, consistency could be defined as

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

where $\models_{\mathsf{LIA}}$ indicates the expression is interpreted in the standard model of linear integer arithmetic.

For a large class of base languages, e.g. those described in [Krogmeier and Madhusudan 2023], the consistency predicate, specialized to any *fixed example*, can be computed by a tree automaton whose size is a function of only $|M|$.

Definition 4.1.3 (Tree Automaton-Computable Semantics)**.** Fix a base language consisting of grammar $G = (S, N, \Delta, P)$, examples $\mathcal{M}$, and predicate $\mathsf{consistent} \subseteq L(G) \times \mathcal{M}$. 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., $L(A_M) = \{e \in L(G) : \mathsf{consistent}(e, M)\}$ and $L(A_{\neg M}) = \{e \in L(G) : \neg\mathsf{consistent}(e, M)\}$.

We refer to A_M and $A_{\neg M}$ in the definition above as *example automata* in forthcoming proofs.

4.2 Solution Concepts for DSL Synthesis

We study DSL synthesis along two dimensions: (1) the mechanism for specifying DSLs over a base language and (2) properties required of a DSL for it to solve the synthesis problem.

Along dimension (1), we consider specifying DSLs using either grammars or (more expressive) macro grammars defined over a 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).

Definition 4.2.1 (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 $N \supseteq N'$, which define productions for new nonterminal symbols $N \setminus N'$. Given a synthesized grammar G, the resulting DSL we consider is given by $\mathsf{extend}(G, G') \coloneqq (S, N, \Delta, P \cup P')$.

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*.

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.

Definition 4.2.2 (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

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

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

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

Definition 4.2.3 (Adequacy)**.** Given a set of learning instances $I_1, \ldots, I_n$, a DSL is *adequate* if for each I_p it contains an expression e such that $\mathsf{solves}(e, I_p)$ holds.

The strong property required of DSLs relates to the *concept ordering* that they induce on the space of expressions. There are different natural concept orderings one can consider, and we describe two of them shortly.

Definition 4.2.4 (Expression complexity)**.** The *expression complexity* for a DSL G is determined by a function $c_G : T_\Delta \to \mathbb{N} \cup \{\infty\}$ which measures the complexity of expressions in G. Note that expressions are not themselves ordered by such functions, e.g., the parse tree depth of distinct expressions e and e' may be equal. Instead, a complexity function for G partitions expressions and orders the cells of that partition.

Definition 4.2.5 (Concept ordering)**.** A *concept ordering* is a relation which compares the complexity of expressions. Given an expression complexity function c_G it defines a binary relation on expressions, which we denote by $\mathsf{order}(e, e', G)$. For example, we may choose $\mathsf{order}(e, e', G) = c_G(e) \leq c_G(e')$.

Given a concept ordering $\mathsf{order}(e, e', G)$, we refer to the stronger property of DSLs as *generalization.* This property implies adequacy, and further requires that the "simplest" expressions, with simplicity measured by the ordering, which solve training examples should also solve testing examples, i.e. they should generalize.

Definition 4.2.6 (Non-generalizing expression)**.** Given an instance $I = (X, Y)$, a *non-generalizing expression* e is one which solves X but does not solve Y.

Example 4.2.1. The regular expression abc fails to generalize for $I = (\{abc^+, abb^-\}, \{abbc^+\})$ because it matches all positive words and no negative words in the training set, but it does not match the positive word in the testing set.

The strong property, i.e. generalization, is defined as follows.

Definition 4.2.7 (Generalization)**.** Given a set of learning instances $I_1, \ldots, I_n$, a DSL G is *generalizing* for a concept ordering $\mathsf{order}(e, e', G)$ if it is adequate (Definition 4.2.3), and additionally, the following holds. For each I_p, there is an expression $e \in G$ such that $\mathsf{solves}(e, I_p)$ holds, and for all $e' \in G$ which are non-generalizing on I_p, we have that $\mathsf{order}(e, e', G)$ holds.

4.3 Depth concept ordering

Useful DSLs organize a space of symbolic expressions in a way that captures domain-specific concepts and computations using "simple" expressions. One common and useful way to measure the simplicity of expressions in DSLs is expression parse tree *depth.* This notion is DSL-dependent, in the sense that an expression e may have small depth in one DSL but only very large depth in another. Formally, depth induces a natural grammar-dependent ordered partition of expressions which captures the preference given to individual expressions by a DSL.

Definition 4.3.1 (Expression depth in a grammar)**.** Given a grammar G, we define the depth complexity $\mathsf{depth}_G : T_\Delta \to \mathbb{N} \cup \{\infty\}$, which for any $e \in G$ is the minimum over all parse trees t for e of the maximum number of nonterminals encountered along any root-to-leaf path in t. If there is no parse tree for e in G then $\mathsf{depth}_G(e) = \infty$.

Example 4.3.1. Consider the following grammars:

$$G_1: \quad S \longrightarrow S + S \ \mid \ x \qquad\qquad G_2: \quad S \longrightarrow x + S \ \mid \ x.$$

The expression $e = x + x + x + x$ has $\mathsf{depth}_{G_1}(e) = 3$ and it has $\mathsf{depth}_{G_2}(e) = 4$.

We define depth concept ordering, which appears later in our results, as follows.

Definition 4.3.2 (Depth concept ordering)**.** Given a grammar G and an expression $e \in T_\Delta$, the *depth concept ordering* is given by $\mathsf{order}(e, e', G) = \mathsf{depth}_G(e) \leq \mathsf{depth}_G(e')$.

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

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

(1) $\mathsf{extend}(G, G')$ is adequate (Definition 4.2.3) and
(2) $\mathsf{enc}(G) \in L(\mathcal{G})$, 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 4.2.3: 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.

For each example $M \in X \cup Y$, e.g. a structure like a Boolean-labeled graph or a pair of input-output values, we assume the existence of a non-deterministic top-down tree automaton A_M whose language is

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

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

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

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

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 tree 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 $t_1, \dots, t_r \in T_\Delta(\{\mathsf{rhs}_{N_i} \ : \ N_i \in N\})$. We use an underscore "_" to describe a default transition when no other case matches.

Mode 1. Find productions. States are drawn from

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

$$\delta(\langle q, \mathbf{start}\rangle, \mathsf{root}) = (\mathbf{down}, \langle q, \mathbf{start}\rangle) \qquad \delta(\langle q, N_i\rangle, \mathsf{lhs}_{N_j}) = (\mathbf{up}, \langle q, N_i\rangle) \vee (\mathbf{right}, \langle q, N_i\rangle)$$

$$\delta(\langle q, \mathbf{start}\rangle, \mathsf{lhs}_{N_i}) = (\mathbf{stay}, \langle q, N_i\rangle) \qquad \delta(\langle q, N_i\rangle, \_) = (\mathbf{up}, \langle q, N_i\rangle) \vee \big(\vee_{(N_i, \alpha) \in P'} (\mathbf{stay}, \langle q, \alpha\rangle)\big)$$

$$\delta(\langle q, N_i\rangle, \mathsf{lhs}_{N_i}) = (\mathbf{up}, \langle q, N_i\rangle) \vee (\mathbf{left}, q) \vee (\mathbf{right}, \langle q, N_i\rangle) \vee \big(\vee_{(N_i, \alpha) \in P'} (\mathbf{stay}, \langle q, \alpha\rangle)\big)$$

Mode 2. Read productions. States drawn from

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

$$\delta(q, x) = \delta_1(q, x) \qquad \delta(\langle q, \mathsf{rhs}_{N_i}\rangle, \_) = (\mathbf{stay}, \langle q, N_i\rangle) \vee \big(\vee_{(N_i, \alpha) \in P'} (\mathbf{stay}, \langle q, \alpha\rangle)\big)$$

$$\delta(q, \mathsf{rhs}_{N_i}) = (\mathbf{stay}, \langle q, N_i\rangle) \qquad \delta(\langle q, f(t_1, \dots, t_r)\rangle, \_) = \mathit{adorn}(t_1, \dots, t_r, \delta_1(q, f))$$

The notation $adorn(t_1, \ldots, t_r, \varphi)$ represents a transition formula obtained by replacing each atom of the form (i, q) in the Boolean formula φ by the atom $(\mathbf{stay}, \langle q, t_i \rangle)$[1].

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

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

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

Proof. Follows easily by construction. See Appendix B.1. □

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 4.1.3). Furthermore, the set of solutions corresponds to a regular set of trees.*

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

$$A \coloneqq A(\mathcal{G}) \times \mathsf{convert}\left(\bigtimes_{p \in [l]} A_{I_p}\right),$$

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

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

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

Corollary 5.3. *Adequate DSL synthesis is decidable in time*

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

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

Remark. Note that the logic of 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. □

In the context of this section, the automaton B_A corresponds to A_I and the automaton A corresponds to the automaton A_1.

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

6 DSL Synthesis

In this section we introduce the DSL synthesis problem for grammars and prove decidability for ordering based on expression depth. Unlike adequate DSL synthesis, this problem asks for a DSL which orders concepts in such a way that small expressions learned on a training set of examples will also generalize to a testing set.

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

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

Input:

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

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

(1) $\mathrm{extend}(G, G')$ is adequate (Definition 4.2.3) and generalizing (Definition 4.2.7) and
(2) $\mathrm{enc}(G) \in L(\mathcal{G})$, 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 turn to proving that DSL synthesis is *decidable* over languages whose semantics can be computed by finite tree automata and when the expression order is given by *parse tree depth*. We start by developing some intuition about *equivalence of grammars*, which underpins the explanation for how an automaton can evaluate arbitrarily large grammars and reason about their induced concept orderings.

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.

Consider what would make two distinct grammars G_1 and G_2 equivalent with respect to $I = (X, Y)$ under the stronger requirement of generalization (Definition 4.2.7). 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. To simplify things, consider how the two grammars might behave over a single example structure M with expressions interpreted over a domain D^M.

Suppose $G_1 = (N_1, \{N_1, N_2\}, \Delta, P_1)$ and $G_2 = (N_1, \{N_1, N_2\}, \Delta, P_2)$, with $\Delta = \{h^1, a^0\}$. Let the domain of the example be $D^M = \{1, 2, 3, 4\}$, with the symbols h and a interpreted as $h^M(1) = 2, h^M(2) = 3, h^M(3) = 4, h^M(4) = 1$, and $a^M = 1$. Suppose G_1 has productions

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

One way to measure the ease with which G_1 expresses properties of M is to consider its expressible properties indexed by depth. A property in this example is a domain value of M that is computed by a term in G_1. Of the 4 domain values for M, some can be computed with shallower terms than others. For any given property we want to know the smallest $d \in \mathbb{N}$ for which it is expressible in G_1 using an expression of depth d but no shallower.

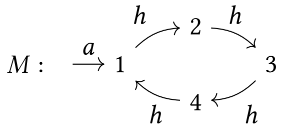
We can encode this information in a behavioral signature for G_1 using a table whose entries are subsets of D and whose columns and rows are indexed by nonterminals and increasing integers, respectively, as depicted in the right half of Figure 2.

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

Now suppose our other grammar G_2 has the following productions

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

If we consider the table for G_2 we find it is identical to the table for G_1. These tables give us a notion of equivalence that captures whether two grammars have the same expressive power, parameterized by *parse tree depth*, over fixed structures. We will use the information in such tables, albeit with a more complex domain D, in the automaton construction to come.



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

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

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

G_3 : $N_1 \leftarrow h^2(a) \mid h^2(N_2)$
$N_2 \leftarrow h^2(N_1)$

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

Fig. 2. (Left) structure M and grammars G_1, G_2. (Middle) behavioral signature for G_1 and G_2 interpreted over M, where entry (i, j) contains values of M reached by $t \in L(N_j)$ of depth at most i. (Right) signature for G_3.

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, for each example $M \in X \cup Y$, we assume the existence of non-deterministic top-down tree automata A_M over alphabet Δ whose language is

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

Additionally, to handle the stronger generalization property of DSL synthesis we also will use a non-deterministic top-down automaton $A_{\neg M}$, whose language is

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

These automata accept the sets of expressions in the base language which are consistent or inconsistent with each example (Definition 4.1.3). We can now define *instance automata* A_1^I and A_2^I:

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

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:

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

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*.

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

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

The notation $[\![t]\!]_R^{A_j}$, for $j \in \{1, 2\}$, denotes the subset of Q_j reachable by running the automaton

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

on term t, where $\delta'_j(q, \mathsf{rhs}_{N_s}) = \mathsf{true}$ 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 $H : \mathcal{P}(Q_1 \sqcup Q_2)^k \to \mathcal{P}(Q_1 \sqcup Q_2)^k$ defined by $H(R) = \langle (H_1(R), \ldots, H_k(R) \rangle$ is monotone with respect to component-wise inclusion of sets, and thus the following sequence converges to a fixpoint after $n \leq k(|Q_1| + |Q_2|)$ steps:

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

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

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

By construction, for the depth concept ordering (Definition 4.3.2) given by

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

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

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

We now have what we need to define the tree automaton A_I whose language is

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

In the following, we summarize the high-level operation of A_I as it relates to recursion tables; a detailed construction including states and transitions can be found in Appendix C.1.

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

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

To achieve the functionality described above, the automaton operates in a few different modes, which we describe below. Recall k is the number of nonterminals. We use D as a shorthand for $\mathcal{P}(Q_1 \sqcup Q_2)$, and so rows of the recursion table are drawn from D^k, and we abuse notation by writing $L \cup L'$ for component-wise union over vectors $L, L' \in D^k$. Besides control information for making transitions between the modes described below, the automaton maintains 3 vectors $L, C, R \in D^k$, where L tracks the component-wise union over all previously constructed rows, C tracks the current row, and R tracks the remaining values which have not yet been obtained.

In **mode 1**, A_I moves to the top of the input tree t. From **mode 1** it enters **mode 2**, in which it guesses which element $C \in D^k$ appears as the next row of $T(t)$, with the requirement that it contain at least one non-empty component, i.e. $C \neq \{\emptyset\}^k$. This non-emptiness requirement is what makes the automaton reject if it constructs the entire table and has not yet accepted. In **mode 3** it traverses the right spine of the tree to verify the guess C. For each nonterminal N_i encountered along the right spine of t, this involves guessing which subset $U \subseteq C_i$ a given production for N_i should reach, given a vector $L \in D^k$ consisting of all previously reached values for all nonterminals. In **mode 4** and **mode 5** it attempts to verify these guesses. In **mode 4**, it simulates the modified instance automaton A'_1 on the right-hand side of a given production to check that all values in U are reachable, assuming those in L are reachable. In **mode 5**, it simulates A'_2 to check that all values in $R_i := (Q_1 \sqcup Q_2) \setminus (L \cup C)_i$ are not reachable, again assuming those in L are reachable. Note that after the automaton reaches the end of the productions on the right spine of t, it simulates **modes 4** and **5** as if the productions P' from the base grammar G' were present in the tree. Finally, it enters **mode 6** to check if the partially guessed column corresponding to the starting nonterminal is already acceptable, and if so it accepts. Otherwise it verifies that no non-generalizing expression has yet been constructed and enters **mode 1** to return to the root of t.

We 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

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

Theorem 6.1. *DSL synthesis is decidable with depth concept ordering (Definition 4.3.2) for any language whose semantics on any fixed structure can be evaluated by tree automata (Definition 4.1.3). Furthermore, the set of solutions corresponds to a regular set of trees.*

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

$$A \coloneqq A(\mathcal{G}) \times \mathsf{convert}\big(\bigtimes_{p \in [l]} A_{I_p}\big),$$

where $\mathsf{convert}(B)$ translates a two-way tree automaton B to a non-deterministic automaton in time $\exp(|B|)$ [Cachat 2002; Vardi 1998]. We have by construction and Lemma C.1 that

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

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

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

Remark. The logic of the construction of A_I, specifically the simulation of A_1^I and A_2^I on the terms appearing in 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 simplificaitons. □

In the context of this section, the automaton C corresponds to A_I, the automaton A corresponds A_1^I, and the automaton B corresponds to A_2^I.

7 DSL Synthesis with Macros

We now introduce variants of the two problems from Sections 5 and 6 which involve spaces of DSLs defined using *macro grammars*. We saw an example of a macro grammar in Section 2.5 and discussed their syntax and semantics in Section 3.1. We establish decidability results for each variant. Due to space constraints, we give only overviews in this section.

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 $I_1, \ldots, I_l$ and meta-grammar $\mathcal{G}$ over $\Gamma(\Delta, N)$
Output: Macro grammar $G = (S, N, \Delta, P)$ such that

(1) $\mathsf{extend}(G, G')$ is adequate (Definition 4.2.3) and
(2) $\mathsf{enc}(G) \in L(\mathcal{G})$, i.e. constraints $\mathcal{G}$ are satisfied

The allowance of macros leads us to a more complex decision procedure. We now explain at a high level the adjustments we use to prove the following result for adequate DSL synthesis with macros. Details are omitted; a complete construction can be found in Appendix D.

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 4.1.3). Furthermore, the set of solutions corresponds to a regular set of trees.*

Proof overview. The proof is similar to that of adequate DSL synthesis from Section 5, except the automaton A_I uses exponentially more states to enable it to deal with macros. In order to simulate the instance automaton A_1 over an input grammar and verify the existence of a single expression solving I, our automaton A_I keeps track of *sets* of distinct expressions which can be generated by a given nonterminal. To understand the reason for this difference with the previous construction, consider the grammar

$$S \rightarrow H(G), \qquad H(1) \rightarrow h(1,1), \qquad G \rightarrow a \mid b.$$

Imagine A_I is reading the production $S \rightarrow H(G)$ and checking that S generates an expression evaluating to some $q \in Q_1$, a state of the automaton A_1. It might check that an expression generated by the productions for H can evaluate to q, *assuming* that the argument G generates an expression which evalutates to some $q_G \in Q_1$. Notice, however, that $H(G) \Longrightarrow h(G,G) \Longrightarrow h(a,b)$ is a valid derivation under outside-in macro semantics, and notice that G generates two distinct expressions, though it was passed as a single argument to H. Thus, in general, the information the automaton needs to remember about the argument G is in fact *the entire set of* values of Q_1 that expressions generated by G can evaluate to.

When evaluating a macro application, the automaton verifies that *sets* of states $Q' \subseteq Q_1$ are achievable. For example, when reading a macro $F(t_1, t_2)$, A_I nondeterministically guesses, for each parameter i, a subset $Q'_i \subseteq Q_1$ that is achieved by t_i. It verifies these guesses by descending into the arguments and recursively checking that t_i can evaluate to Q'_i, while simultaneously passing values Q'_i to all the productions of F. It then verifies that the F productions together can produce the needed values Q' given the assumption about the arguments. Besides this aspect of handling 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 decidability for depth ordering. The argument shares much of the structure of Section 6.2, but is significantly more complex, owing to the interaction between depth and nested macros.

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

Input: Instances $I_1, \ldots, I_l$ and meta-grammar $\mathcal{G}$ over $\Gamma(\Delta, N)$
Output: Macro grammar $G = (S, N, \Delta, P)$ such that:

(1) $\mathsf{extend}(G, G')$ is adequate (Definition 4.2.3) and generalizing (Definition 4.2.7) and
(2) $\mathsf{enc}(G) \in L(\mathcal{G})$, i.e. constraints $\mathcal{G}$ are satisfied

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 4.1.3).*

The proof follows the structure of Section 6.3. The main difference is that the automaton A_I, in order to accurately compute the values achievable by nonterminals at a given depth, keeps track of all previously computed rows of the recursion table individually, rather than merely keeping track of the componentwise union of previously computed rows. This finer-grained information about depth allows the table cells to be updated accurately for rules involving nested macros.

8 Related Work

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

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

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

Library learning. Recent work 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 small 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 we study program and abstraction learning simultaneously, whereas in library learning the programs are given as input. Closer in spirit to our work are the EC^2 and DreamCoder systems of [Ellis et al. 2018, 2023], which learn and compress a library of subroutines in tandem with a search (guided by a neural network) over programs in the library to solve program synthesis problems. In contrast to our work, these systems do not give formal guarantees about how well abstractions generalize, and they cannot declare there is *no library* over a given signature which solves a set of learning problems.

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

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

9 Conclusion

We formulated the problem of synthesizing DSLs from few-shot learning instances. Synthesized DSLs are expressed as (macro) grammars, which are defined over a base language which gives semantics to the DSL. Each input learning instance is split into training and testing sets of examples. The notion of expression complexity, which varies as a parameter, induces an ordered partition of the expressions in the language for a given DSL. The problem asks for a grammar which is expressive enough to contain solutions for each input learning instance, while ensuring that for each one, some of the least complex expressions according to the ordering that are consistent with the training set are also consistent with the testing set. Intuitively, the DSL synthesis problem asks for a DSL such that picking the simplest concepts consistent with training data, results in expressions which generalize, in the sense that they are also consistent with testing data.

We proved that DSL synthesis is decidable when expression complexity is given by parse tree depth 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 a variant with the weaker requirement that DSLs must only express some solution for each instance. Finally, we proved decidability for extensions of the weak and strong variants to the case where grammars can use macros.

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

Acknowledgments

10 Data Availability Statement

This paper has no accompanying artifact as the contributions are theoretical.

References

Rajeev Alur, Rastislav Bodík, Eric Dallal, 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

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

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

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

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

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

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

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

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

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

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

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

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

H. W. Kamp. 1968. *Tense logic and the theory of linear order.* Ph.D. Dissertation. University of California, Los Angeles.

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

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

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

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

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

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

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

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

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

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

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

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

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

Amir Pnueli. 1977. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. 46–57. https://doi.org/10.1109/SFCS.1977.32

Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: a framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 107–126. https://doi.org/10.1145/2814270.2814310

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

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

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

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

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

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

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

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

A Section 3: Encoding grammars and decoding grammar trees

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

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

The grammar G corresponding to a grammar tree t of the form $\mathsf{root}(\mathsf{lhs}_{N_i}(x, y))$ over alphabet $\Gamma(\Delta, N)$, is given by $\mathsf{dec}(t) = (N_i, N, \Delta, \langle (N_i, \mathsf{dec}_t(x)), \mathsf{dec}_p(y) \rangle)$, where dec_t and dec_p are computed recursively as follows.

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

B Section 5: Proof of Decidability

B.1 Proof of Lemma 5.1

We use the notion of a run for two-way alternating tree automata from [Cachat 2002]. Though our presentation in the main text used terms over ranked alphabets, here it is more convenient to use the language of labeled trees. Let W be a set of directions using which our terms over $\Gamma(\Delta, N)$ can be described as finite $\Gamma(\Delta, N)$-labeled W-trees (T, l). Let $A_X = (Q, \Gamma(\Delta, N), I, \delta)$ and $A_1 = (Q_1, \Delta, I_1, \delta_1)$.

First we argue that $L(A_X) \subseteq \{t \in T_{\Gamma(\Delta,N)} : \mathsf{solves}(\mathsf{extend}(\mathsf{dec}(t), G'), X)\}$. Suppose $t \in L(A_X)$, and let the corresponding W-tree for t be (T, l). Let (T_r, r) be an accepting run of A_X on t, which in our case means that each branch is finite, and where T_r is a $(W^* \times Q)$-labeled Z tree, for a suitable set Z, and $r : T_r \longrightarrow (W^* \times Q)$ labels each node of the run with (p, q), whose meaning is that the automaton passes through input tree node p in state q.

The main observation is that there is a subtree of the run (T_r, r) that encodes an expression $e \in L(\mathsf{dec}(t))$ on which the instance automaton A_1 accepts, and thus for which $\mathsf{solves}(e, X)$ holds. There may be multiple such subtrees (which possibly overlap) and thus there may be multiple such expressions encoded in (T_r, r). Such an expression can be constructed from the run by following any branch from a node labeled by (w, q) with $l(w) = \mathsf{lhs}_{N_i}$ or $l(w) = \mathsf{root}$. For nodes labeled by (w, q) with $l(w) = x$, for $x \in \Delta \sqcup \{\mathsf{rhs}_{N_i} : N_i \in N\}$ with $q \in Q_1$, there is a subset of successors in T_r which can be followed to construct the subterms for the symbol x, possibly multiple successors in the case of $x \in \Delta^{>0}$, or a single successor in the case of $x \in \{\mathsf{rhs}_{N_i} : N_i \in N\}$. For nodes labeled by $(w, (q_1, \alpha))$ with $(q_1, \alpha) \in Q_1 \times \mathit{subterms}(P')$, a similar procedure can construct the expression from α by following any branch at nodes labeled by $\{\mathsf{rhs}_{N_i} : N_i \in N\}$.

Next we argue that $L(A_X) \supseteq \{t \in T_{\Gamma(\Delta,N)} : \mathsf{solves}(\mathsf{extend}(\mathsf{dec}(t), G'), X)\}$. Suppose t is such that $\mathsf{solves}(\mathsf{extend}(\mathsf{dec}(t), G'), X)\}$ holds. Let $N_i \in N$ be the topmost nonterminal in the spine of productions for t, which is the starting nonterminal for $\mathsf{dec}(t)$ and also $\mathsf{extend}(\mathsf{dec}(t), G')$, and thus there exists $e \in L(N_i)$ within $\mathsf{extend}(\mathsf{dec}(t), G')$ such that $\mathsf{solves}(e, X)$. Now, it follows that A_1 accepts e. Since $e \in L(N_i)$ within $\mathsf{extend}(\mathsf{dec}(t), G')$, the productions used in a parse e_p for e

must appear in t for some production in the spine, or else they are from G', in which case they appear in the states of A_X. Given the parse e_p, we can straightforwardly construct a finite run $r(e_p)$ for A_X on t. This run satisfies the transitions because e_p uses the productions appearing in t as well as those in G' and it satisfies the simulation of A_1 because A_1 accepts e.

C Section 6: Proof of Decidability

C.1 Construction of A_I

We define a two-way alternating tree automaton $A_I = (Q, \Gamma(\Delta, N), Q_i, \delta)$ with acceptance defined by the existence of a finite run satisfying the transition formulas. It follows the logic described in the overview from the main text.

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

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

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

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

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

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

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

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

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

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

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

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

Transitions for **M4a**:

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

Transitions for **M4b**:

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

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

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

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

Transitions for **M5a**:

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

Transitions for **M5b**:

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

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

[3] We assume directions in δ_1 and δ_2 are the numbers $1(\mathbf{left})$, $2(\mathbf{right})$, $3\ldots$, etc.

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

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

- $\delta(\langle U, \mathbf{solve}\rangle, \_) = \text{if } U \cap Q_1^i \neq \emptyset \text{ then true else false}$
- $\delta(\langle U, \mathbf{ok}\rangle, \_) = \text{if } U \cap Q_2^i \neq \emptyset \text{ then false else true}$

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

$$Q := \mathbf{M1} \sqcup \mathbf{M2} \sqcup \mathbf{M3} \sqcup \mathbf{M4} \sqcup \mathbf{M5} \sqcup \mathbf{M6} \qquad \text{and} \qquad Q_i := \{\langle \emptyset, \emptyset, \{Q_1 \sqcup Q_2\}^k, \mathbf{reset}\rangle\}.$$

By construction, we have the following.

Lemma C.1. $L(A_I) = \{t \in T_{\Gamma(\Delta,N)} \ : \ \mathsf{solves}(\mathsf{extend}(\mathsf{dec}(t), G'), I)\}$.

Proof. Appendix C.2. □

C.2 Proof of Lemma C.1

We use the notion of a run for two-way alternating tree automata from [Cachat 2002]. Though our presentation in the main text used terms over ranked alphabets, here it is more convenient to use the language of labeled trees. Let W be a set of directions using which our terms over $\Gamma(\Delta, N)$ can be described as finite $\Gamma(\Delta, N)$-labeled W-trees (T, l). Let $A_I = (Q, \Gamma(\Delta, N), I, \delta)$ and $A_1 = (Q_1, \Delta, F_1, \delta_1)$ and $A_2 = (Q_2, \Delta, F_2, \delta_2)$.

First we argue that $L(A_I) \subseteq \{t \in T_{\Gamma(\Delta,N)} \ : \ \mathsf{solves}(\mathsf{extend}(\mathsf{dec}(t), G'), I)\}$. Suppose $t \in L(A_I)$, and let the corresponding W-tree for t be (T, l). Let (T_r, r) be an accepting run of A_I on t, which in our case means that each branch is finite, and where T_r is a $(W^* \times Q)$-labeled Z tree, for a suitable set Z, and $r : T_r \longrightarrow (W^* \times Q)$ labels each node of the run with (p, q), whose meaning is that the automaton passes through input tree node p in state q.

The main observation is that from the run (T_r, r) we can construct a recursion table $T(\mathsf{extend}(\mathsf{dec}(t), G'))$ whose column corresponding to the starting nonterminal is acceptable. This follows straightforwardly from the construction of A_I, though it is tedious. We sketch the argument now. The rows of the recursion table are the vectors C in the state component labeling the run at specific nodes. Along any path in T_r, at the ith node at which a **mode 1** state is entered (starting from the root of T_r), including at the beginning of the run when $i = 1$, it is the case that the ith row of $T(\mathsf{extend}(\mathsf{dec}(t), G'))[i :]$ is precisely the vector C. It is the case also that the automaton state at that node has a vector L which is the componentwise union over all previous vectors L at earlier positions in the run. Let $Z_0 = \langle \emptyset, \ldots, \emptyset \rangle \in \{\emptyset\}^k$, where $k = |N|$. We claim that the state component C satisfies $C = H^{i-1}(Z_0) \setminus H^{i-2}(Z_0)$ and L satisfies $L = H^{i-2}(Z_0)$, for $i \geq 2$, where set operations are extended to vectors by acting componentwise. The property for L is preserved by transitions in **mode 2**, where the new value of L is the union of C with the previous value of L, i.e., L gets the value $H^{i-2}(Z_0) \cup (H^{i-1}(Z_0) \setminus H^{i-2}(Z_0)) = H^{i-1}(Z_0)$. The property for C is preserved by transitions of **mode 3**, **mode 4**, and **mode 5**. In **mode 3**, it is ensured that C is drawn from remaining values in $(\{Q_1 \sqcup Q_2\}^k \setminus H^{i-2}(Z_0)$ and also that all of C is in fact reachable, as the run cannot pass through the node labeled end without having verified that all of C is computable either by existing productions in the tree t or by productions from G', which are memorized in the states. In **mode 3**, it is also ensured that the first column of $T(\mathsf{extend}(\mathsf{dec}(t), G'))$ is acceptable by checking that the current row C (copied in W) has as its first component, either a set of values containing an initial state of A_1, or if not, a set of values disjoint from the initial states of A_2. In **mode 4** and **mode 5**, it is ensured that the entries of C are in fact generated by the productions available and that no values excluded from C can be generated (thus ensuring that the guess for C in fact contains all of $H^{i-1}(Z_0) \setminus H^{i-1}(Z_0)$ rather than a strict subset. Finally, since T_r is finite and satisfies the transitions, any time it returns to the root of t we know that no non-generalizing expression has been found, and furthermore, the run cannot avoid ending in **mode 6** in a **solve** state, which ensures that the first column of $T(\mathsf{extend}(\mathsf{dec}(t), G'))$ is acceptable.

Now we argue that $L(A_I) \supseteq \{t \in T_{\Gamma(\Delta,N)} : \mathsf{solves}(\mathsf{extend}(\mathsf{dec}(t), G'), I)\}$. Suppose t is such that $\mathsf{solves}(\mathsf{extend}(\mathsf{dec}(t), G'), I)\}$ holds. Let $N_i \in N$ be the topmost nonterminal in the spine of productions for t, which is the starting nonterminal for $\mathsf{dec}(t)$ and also $\mathsf{extend}(\mathsf{dec}(t), G')$. Thus there exists $e \in L(N_i)$ within $\mathsf{extend}(\mathsf{dec}(t), G')$ such that $\mathsf{solves}(e, I)$ and for all non-generalizing $e' \in L(N_i)$ within $\mathsf{extend}(\mathsf{dec}(t), G')$, we have that $\mathsf{depth}(e, \mathsf{extend}(\mathsf{dec}(t), G')) \leq \mathsf{depth}(e', \mathsf{extend}(\mathsf{dec}(t), G'))$. We can construct a run r_t by making choices according to the recursion table $T(\mathsf{extend}(\mathsf{dec}(t), G'))$. The run will only construct the table up to stage i, where i is the depth of a shallowest parse tree e_p for e within $\mathsf{extend}(\mathsf{dec}(t), G')$, which by assumption is at least as shallow as any parse tree for non-generalizing $e' \in L(N_i)$ within $\mathsf{extend}(\mathsf{dec}(t), G')$. Thus the run can satisfy the checks of **mode 6** within i passes through a the root node of t.

D Section 7: Adequate DSL synthesis with macros

We describe an automaton A_X which accepts an input tree if it encodes a macro grammar which contains a solution for the set of examples X. Because a macro can copy its input parameters, which may, under outside-in semantics, be arbitrary terms involving other macros, our automaton will keep track of sets of states which the parameters may evaluate to.

Construction. Suppose $A_1 = (Q_1, \Delta, I_1, \delta_1)$ is an automaton accepting all expressions which satisfy the examples X. We define a two-way alternating tree automaton $A_X = (Q, \Gamma(\Delta, N), I, \delta)$. The automaton operates in two modes, as before. 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. Because nonterminals can be nested, these two modes can be entered in a single transition.

Below we use $N_i \in N^{=0}$, $F \in N^{>0}$, $Y, Y' \in N$ with $Y \neq N_i$, $Y' \neq F$, $q_i \in I_1$, $C \subseteq Q_1$, $x, f \in \Delta$, and $t_1, \ldots, t_r \in T_\Delta(\{\mathsf{rhs}_{N_i} : N_i \in N^{=0}\})$. We use an underscore "_" to describe a default transition when no other case matches. Let $a^* = \max(\{\mathsf{arity}(X) : X \in N\})$.

Mode 1. Find productions. States are drawn from

$$\mathbf{M1} \coloneqq \big(\mathcal{P}(Q_1) \times \{\mathbf{start}\}\big) \cup \big(\mathcal{P}(Q_1) \times N\big) \cup_{i \in [a^*]} \big(\mathcal{P}(Q_1)^i \times \mathcal{P}(Q_1) \times N^{=i}\big).$$

$$\begin{aligned}
\delta(\langle C, \mathbf{start}\rangle, \mathsf{root}) &= (\mathbf{down}, \langle C, \mathbf{start}\rangle) \\
\delta(\langle C, \mathbf{start}\rangle, \mathsf{lhs}_{N_i}) &= (\mathbf{stay}, \langle C, N_i\rangle) \\
\delta(\langle C, N_i\rangle, \mathsf{lhs}_Y) &= \vee_{\{(C_1,C_2):C_1\cup C_2=C\}} (\mathbf{up}, \langle C_1, N_i\rangle) \vee (\mathbf{right}, \langle C_2, N_i\rangle) \\
\delta(\langle C, N_i\rangle, \mathsf{lhs}_{N_i}) &= \vee_{\{(C_1,C_2,C_3,C_4):C_1\cup C_2\cup C_3\cup C_4=C\}} (\mathbf{up}, \langle C_1, N_i\rangle) \wedge (\mathbf{left}, C_2) \wedge (\mathbf{right}, \langle C_3, N_i\rangle) \\
&\quad \wedge_{q\in C_4} \big(\vee_{(N_i,\alpha)\in P'} (\mathbf{stay}, \langle \{q\}, \alpha\rangle)\big) \\
\delta(\langle C, N_i\rangle, \_) &= \vee_{\{(C_1,C_2):C_1\cup C_2=C\}} (\mathbf{up}, \langle C_1, N_i\rangle) \wedge_{q\in C_2} \big(\vee_{(N_i,\alpha)\in P'} (\mathbf{stay}, \langle \{q\}, \alpha\rangle)\big) \\
\delta(\langle C_1, \ldots, C_k, C, F\rangle, \mathsf{lhs}_{Y'}) &= \vee_{\{(C_1',C_2'):C_1'\cup C_2'=C\}} (\mathbf{up}, \langle C_1, \ldots, C_k, C_1', F\rangle) \wedge (\mathbf{right}, \langle C_1, \ldots, C_k, C_2', F\rangle) \\
\delta(\langle C_1, \ldots, C_k, C, F\rangle, \mathsf{lhs}_F) &= \vee_{\{(C_1',C_2',C_3'):C_1'\cup C_2'\cup C_3'=C\}} \\
&\quad (\mathbf{up}, \langle C_1, \ldots, C_k, C_1', F\rangle) \\
&\quad \wedge (\mathbf{left}, \langle C_1, \ldots, C_k, C_2', F\rangle) \wedge (\mathbf{right}, \langle C_1, \ldots, C_k, C_3', F\rangle) \\
\delta(\langle C_1, \ldots, C_k, C, F\rangle, \_) &= (\mathbf{up}, \langle C_1, \ldots, C_k, C, F\rangle) \text{ // base language is macro-less}
\end{aligned}$$

Mode 2. Read productions. States drawn from

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

$$\begin{aligned}
\delta(C, x) &= \wedge_{q \in C}\, \delta_1(q, x) \\
\delta(\langle C, f(t_1, \ldots, t_r)\rangle, \_) &= \mathit{adorn}(t_1, \ldots, t_r, \wedge_{q \in C}\, \delta_1(q, f)) \\
\delta(C, \mathsf{rhs}_{N_i}) &= (\mathbf{stay}, \langle C, N_i \rangle) \\
\delta(\langle C, \mathsf{rhs}_{N_i}\rangle, \_) &= \vee_{\{(C_1,C_2) : C_1 \cup C_2 = C\}}\ (\mathbf{stay}, \langle C_1, N_i \rangle) \wedge \big(\wedge_{q \in C_2}\ \big(\vee_{(N_i,\,\alpha) \in P'} (\mathbf{stay}, \langle \{q\}, \alpha \rangle)\big)\big) \\
\delta(C, \mathsf{rhs}_F) &= \vee_{C_1, \ldots, C_k \in \mathcal{P}(Q_1)}\ \big(\wedge_{i \in [k]} (i, C_i)\big) \wedge (\mathbf{up}, \langle C_1, \ldots, C_k, C, F \rangle)
\end{aligned}$$

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

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

$$Q \coloneqq \mathbf{M1} \cup \mathbf{M2}, \qquad I = \{\langle \{q_i\}, \mathbf{start} \rangle \ :\ q_i \in I_1\} \subseteq \mathbf{M1}.$$

We have the following by construction.

Lemma D.1. $L(A_X) = \big\{t \in T_{\Gamma(\Delta,N)} \ :\ \mathsf{solves}(\mathsf{extend}(\mathsf{dec}(t), G'), X)\big\}$.

The rest of the proof is similar to that of adequate DSL synthesis for grammars and gives us:

Theorem D.2. *Adequate DSL synthesis with macros is decidable for any language whose semantics over fixed structures can be evaluated by tree automata. Furthermore, the set of solutions corresponds to a regular set of trees.*

The size of the two-way automaton A_X from this section is exponential in the size of A_1, giving us the following, similar to before.

Corollary D.3. *For the languages covered in Theorem D.2, adequate DSL synthesis with macros is decidable in time* $\mathrm{poly}(|\mathcal{G}|) \cdot \exp(l \cdot \exp(m))$, *where l is the number of instances and m is the maximum size over all instance automata.*

E Section 7: DSL synthesis with macros

Proof overview. Similar to Section 6.3, we construct an automaton A_I which accepts an input tree if it encodes a macro grammar which solves an instance $I = (X, Y)$. Because macros enable nonterminals to be nested arbitrarily deeply, and because this problem requires paying attention to the depth of expressions, the automaton A_I uses many more states than the corresponding construction from Section 6.3. It shares much of the structure related to recursion tables, but in building up each row one after the other, it must keep fine-grained information about all previous rows to handle nested macros. Additionally, the entries of the recursion table corresponding to macro symbols indicate *functions* from tuples of columns of the recursion table to a set of values achieved by the macro symbol with a given depth budget. In fact, the entries for macro symbols indicate functions from tuples of sets of values to the *new values* that are achievable given higher depth budget (as in the construction of Section 6.3).

The main complication, as mentioned above, is that to keep track of depth in the presence of macros, we need the automaton to make a distinction between values achieved at different previous depths, as opposed to lumping them together as a set of values achievable in depth less than some bound. The way this can be handled is by allowing the automaton to encode in its states *all* rows of the recursion table up to each individual depth. Though this is quite complex, it is sufficient to implement the same protocol for nondeterministically guessing and verifying the rows of the recursion table for a macro grammar.