Synthesizing DSLs for Few-Shot Learning

ANONYMOUS AUTHOR(S)

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 novel DSL synthesis problem we introduce asks for a grammar over the base language that guarantees that small expressions solving training samples also solve corresponding testing samples. We prove that the problem is decidable for a class of languages whose semantics over fixed structures can be evaluated by tree automata and when expression size corresponds to parse tree depth in the grammar, and, furthermore, the grammars solving the problem correspond to a regular set of trees. We also prove decidability results for variants of the problem where DSLs are only required to express solutions for input learning problems and where DSLs are defined using macro grammars.

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].

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.

2025. ACM XXXX-XXXX/2025/3-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 can be used to effectively solve each of the problems in $\mathcal{I}$. Each $p \in \mathcal{I}$ is itself a learning problem: it includes a set of training examples X_p and a set of testing examples Y_p. We require 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 described by $\mathcal{G}$.

Intuitively, the second property 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. This property is a significant, novel contribution of our problem formulation. It introduces a new signal for learning DSLs which is not part of existing related problems such as library learning [Bowers et al. 2023; Cao et al. 2023], or the problem addressed by systems like DreamCoder [Ellis et al. 2023], where the goal is to refactor an existing DSL to favor useful concepts. We introduce an *inexpressivity signal*, which says that certain concepts should not be expressible, or if they are expressible, they should not be expressed succinctly. In addition to expressing useful concepts, we wish to find a language specific to the domain, i.e., a DSL, which may be less expressive than the base language we begin with.

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.

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

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

Problem 3: DSL synthesis. Is there a DSL $\mathcal{H}$ satisfying constraints $\mathcal{G}$ such that, for every instance $p \in I$, there are concepts in $\mathcal{H}$ that satisfy the training set X_p, and the smallest ones, according to the concept ordering, also satisfy the testing set Y_p. If so, construct the DSL.

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

The adequate DSL synthesis problem is the first 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 in the setting where DSLs are defined using 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 DSL synthesis.

The first reason for the increase in complexity is that we design tree automata which read trees that encode DSLs, and these automata must verify existence of arbitrarily large solutions expressed in these DSLs. Witnessing the non-existence of solutions involves, in general, examining all expression derivations within an encoded DSL.

The second, more interesting 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 leveraging 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, for *any* DSL of arbitrary size, the table of results computed by this dynamic programming algorithm is essentially finite, given a set of learning instances. The contents of cells in the table come from a finite domain for any fixed instance, and thus the table rows repeat at a certain point. Consequently, we can translate the dynamic programming algorithm into 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: Motivation and Examples

In this section we motivate the *domain-specific language 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. Such *domain-specific* languages allow expressing properties *common* in the domain using *small/-succinct* expressions and perhaps disallow or make complex the expression of irrelevant concepts.

In this paper, the DSLs that we aim to identify are those that facilitate *few shot learning*. More precisely, we aim to find DSLs that can succinctly express solutions to typical few shot learning problems in a domain. In particular, this enables *synthesis* algorithms to solve few shot learning instances by searching for and identifying the smallest expressions in the DSL that satisfy the examples, and in that way identify the correct concept in the domain.

As explained in the Introduction, the DSL synthesis problem asks that the smallest solutions expressible in the DSL that solve the training examples also solve the testing examples, for each of the few shot instances given.

Since DSL design is crucial in few shot learning, in a variety of synthesis settings experts have designed DSLs for example-based expression synthesis by examining typical domain concepts and finding ways to express them succinctly. They then develop algorithms that search for the shortest expression to solve a few-shot learning instance. In this work, we are interested in the automation of the DSL design task, and specifically we study learning the DSL from a class of few shot learning instances.

We discuss examples of DSL design for synthesis below, and explain how they fit our problem definition, including the grammar and macro grammar DSL synthesis problems we introduce and prove decidability results for.

2.1 DSLs for Program Synthesis: Spreadsheet Programming, Etc.

The FlashFill system [Cambronero et al. 2023; Gulwani 2011] uses a bespoke DSL for expressing commonly used character string transformation programs for Excel spreadsheets. The paper in fact talks about the careful design of the DSL, and in particular why a general purpose language (like say Python) would not be useful, arguing that the search would become too complex and *"more importantly, allow the large number of functions to be combined in unintuitive ways to produce undesirable programs"*. Core features of that DSL include an operation $\mathsf{SubStr}(s, P, P)$ for extracting substrings from a string s in an input spreadsheet cell, where P is a nonterminal that generates positions, an operation $\mathsf{Pos}(R, R, C)$, which generates a position defined as the cth one whose substring to the left matches a regular expression $r_1 \in L(R)$ and whose substring to the right matches a regular expression $r_2 \in L(R)$.

In our formulation, we seek automatic design of such DSLs given sample problem instances, and furthermore, the domain-specific operations would be defined using *macros*. For instance, starting from a generic programming language with recursion, the $\mathsf{SubStr}(s, P, P)$ macro could be defined using code which recurses over the input s in order to find the left position of the substring and return the string ending at the right position.[1]

The literature is replete with design of DSLs for various program synthesis tasks, too numerous to list— a few examples are the small functional programming language on lists defined for the Sketch-n-Sketch framework for SVG manipulation [Chugh et al. 2016], DSLs that use an algebra of data extracting operators to extract structured data from text using examples [Le and Gulwani 2014], DSLs for synthesizing barriers for crash consistency of file systems [Bornholt et al. 2016], and DSLs for abstract queries used to synthesize SQL queries from examples [Wang et al. 2017a]. Our work aims to automate the construction of such DSLs given sample synthesis problems in such domains.

[1]Note that, while our problem is motivated by synthesis in domains such as spreadsheet programming, there are often other facets to DSL design that we do not consider. In particular, the DSL used in FlashFill has other properties, such as using effectively invertible top operators and effectively enumerable bottom operators, which facilitate *faster* search algorithms. Such "search performance" requirements of DSLs are out of scope of this paper.

2.2 DSL-supporting synthesis tools

The use of DSLs in program synthesis is so ubiquitous that there are several tools that provide explicit support for defining DSLs, with their semantics, in order to facilitate synthesis.

The Rosette framework [Torlak and Bodik 2013] and the Grisette framework [Lu and Bodík 2023] allow definition of solver-aided DSLs for synthesis, with user-defined syntax and semantics, and have facilitated several synthesis projects. The syntax-guided synthesis (SyGuS) framework supports DSLs with user-specified grammars that define syntax (semantics is often fixed a priori) [Alur et al. 2015]. Recent approaches such as SemGuS extend the framework to user-specified DSL semantics [Kim et al. 2021]. In the context of our work, example-based synthesis in the SyGuS formalism could be targeted by our formulation of DSL synthesis with grammars, while synthesis for more complex DSLs with semantics defined by new functions could be targeted by our formulation of DSL synthesis with macro grammars.

2.3 Synthesizing Invariants; Feature Engineering

Symbolic concept learning, including techniques for learning decision trees, Boolean concepts, and symbolic regression, often requires a set base features. 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 designed by hand.

As an example, consider the GPUVerify tool [Betts et al. 2012], which leveraged hand-crafted rules for generating basic candidate invariants for GPU kernels. This set was then used to compute the strongest conjunction over the base invariants using the Houdini algorithm [Flanagan and Leino 2001]. Many hand-crafted rules were found useful, e.g., if $i := i \times 2$ occurs in a loop body, then predicates expressing the loop index i is less than a power of 2 can be useful, e.g., $i < 2$, $i < 4$, $i < 8$, etc., which are common for tree reduction computations. Or, when threads access fixed-size contiguous chunks of a shared array, useful predicates express that the write index is within a bounded region of the array that depends on thread identifier, e.g., $id \times c \leq write\_idx$ and $write\_idx < (id + 1) \times c$, where id, c, and $write\_idx$ are, respectively local thread identifiers, constant offsets, and indices where writing occurs in the array. Only a subset of rules was used for any given invariant inference problem, since most rules are only useful when specific, statically detectable patterns occur in the kernels. In contrast to these hand-crafted predicates, DSL synthesis would seek to automatically discover them from instances of invariant learning problems, which could be sampled from a benchmark of programs to verify. In this setting, we could use a meta-grammar $\mathcal{G}$ to encode the constraint that invariants are *conjunctions* over a set of base predicates, with a DSL synthesis algorithm tasked with determining a good set of base predicates.

DSL synthesis, as proposed in this paper, can serve as a formulation of the feature synthesis problem, with the goal of discovering domain-specific symbolic features using few-shot learning instances drawn from a domain. We can formulate the question as follows: is there a set of n features, each drawn from a class of functions over some given low-level 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 sampled from a domain? Such engineered features can then be used in downstream symbolic learning algorithms for program synthesis [Alur et al. 2015] or symbolic regression [La Cava et al. 2021].

2.4 Library Learning

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. We also introduce the problem of capturing the domain precisely in a DSL, which may be *less expressive* than the base language against which it is defined. Furthermore, we prove decidability results for a class of DSL synthesis problems while earlier work does not provide any decidability results.

3 Preliminaries

3.1 Alphabets, Tree Automata, and Tree Macro Grammars

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

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

Definition 3.1.2 (Tree Macro Grammar)**.** A tree macro grammar[2], or simply *macro grammar*, adds parameters to nonterminal symbols of a regular tree grammar. It 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 rules drawn from $N \times T_{\Delta\cup N}(\mathbb{N})$. We often write rules (N, t) as $N \to t$ and indicate several rules as usual by $N \to t_1 \mid \cdots \mid t_k$.

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* as a special case.

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

Macro grammar G

$$
\begin{array}{lcl}
N_1 & \rightarrow & N_3(N_2) \\
N_2 & \rightarrow & h(N_1) \mid a \\
N_3(1) & \rightarrow & g(1,1)
\end{array}
$$

Encoding $\mathsf{enc}(G) \in T_{\Gamma(\Delta,N)}$

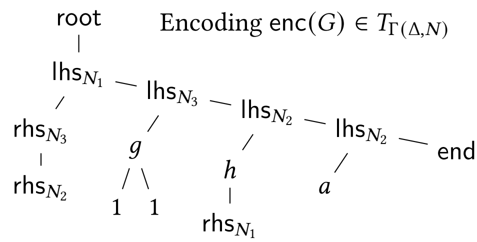

Fig. 1. (Left) Macro grammar G over $\Delta = \{a^0, h^1, g^2\}$ and nonterminals $N = \{N_1^0, N_2^0, N_3^1\}$ and (Right) its encoding as a tree $\mathsf{enc}(G) \in T_{\Gamma(\Delta,N)}$ over grammar alphabet $\Gamma(\Delta, N)$.

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

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

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

Example 3.1.1. Consider the macro grammar defined by rules

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

where integers indicate the parameters for a macro symbol. Observe that we could apply productions to "outermost" macro symbols first, as in $S \Longrightarrow F(H) \Longrightarrow f(H,H) \Longrightarrow f(a,b)$, 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.

Example 3.1.1 illustrates a choice in how to define *derivations* for macro grammars, which in general affects their expressive power [Comon et al. 2007; Fischer 1968]. We will only consider "outermost" derivations, but our techniques and results apply to other choices as well. *Outermost* derivations are those in which rules are never applied to rewrite a nonterminal M if it appears as a subterm of another nonterminal N. This can be formalized using *contexts* (e.g. see [Comon et al. 2007, Chapters 2.1 and 2.5]) by adding the requirement that any context used in defining the derivation relation must not contain nonterminal symbols.

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

3.2 Encoding Grammars as Trees

We will define tree automata whose inputs are trees that encode grammars. Figure 1 shows an example of how we choose to encode a grammar as a tree; we arrange the grammar rules along the topmost right-going spine of the tree and use symbols lhs_{N_i} and rhs_{N_i} to distinguish between occurrences of nonterminal N_i on the left-hand and right-hand sides of rules. We use positive integers to indicate the parameters for macro symbols. We write $\Gamma(\Delta, N)$ to denote *grammar alphabets.*

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

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

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

4 Formulating DSL Synthesis

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

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

4.1 Learning Instances: Expressive Power and Relative Succinctness

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

Definition 4.1.1 (Learning Instance)**.** A *learning instance* is a pair (X, Y) consisting of a set X of *training* examples and a set Y of *testing* examples.

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

Expressivity. The first signal says that, given a learning instance we would like to solve, an adequate DSL must have enough power to express *some* concept which solves it[3]. That is, we want a DSL which contains some expression that satisfies all the examples $X \cup Y$. Consider a DSL for the situation in Figure 2 whose language of expressions consists of Boolean combinations of a fixed set of basic predicates capturing rectangles in the plane, e.g. $r_i \coloneqq (1 < x < 3) \wedge (2 < y < 4)$. Any DSL containing an expression equivalent to $\varphi \coloneqq r_3 \wedge r_4 \wedge r_5$ (shown in Figure 2) is adequate because φ satisfies all training examples X (positive points P) and also all the testing examples Y (negative points N). Note that the expressivity signal treats a learning instance as a set of examples $X \cup Y$ and thus forgets about the distinction between training and testing examples.

Relative Succinctness and Inexpressivity. In addition to expressing relevant concepts, a DSL should also *precisely* capture those relevant concepts and perhaps little or nothing else. DSLs need not be fully expressive, and it is in fact a feature if they avoid expressing irrelevant information

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

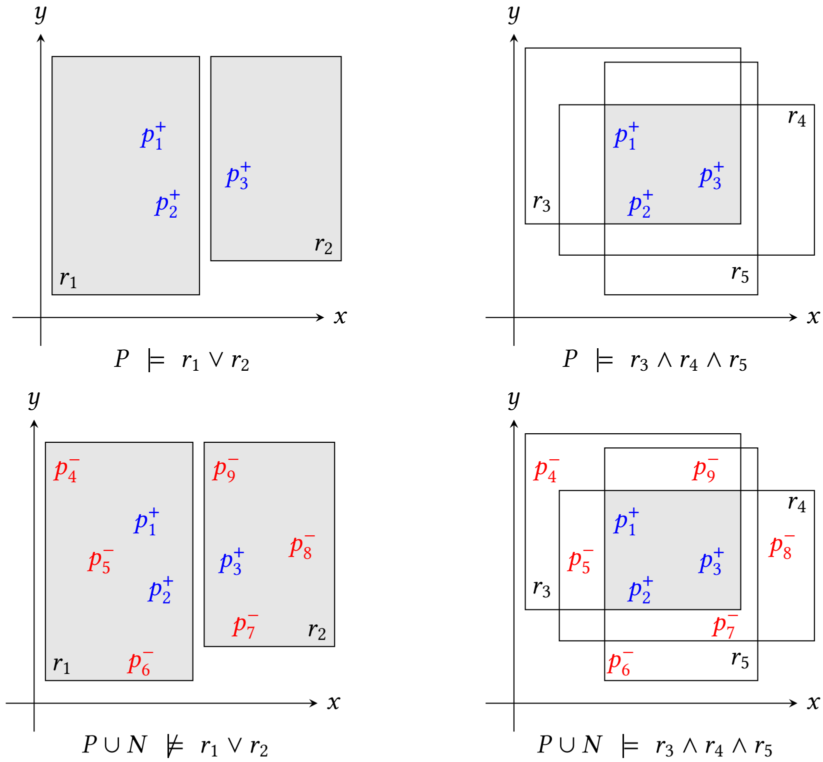

Fig. 2. A learning instance $I = (P, N)$ where examples are points in the plane. Training examples P are the positive points (in blue) and testing examples N are the negative points (in red). Top: the training examples are satisfied by both $r_1 \vee r_2$ and $r_3 \wedge r_4 \wedge r_5$, with r_i signifying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

which does not reflect the domain in question[4]. To address this aspect of DSLs, we formulate a *relative succinctness* constraint which requires that specific concepts should be expressed *less succinctly than other concepts (or not all)* in a synthesized DSL.

To understand this inexpressivity signal, let us consider Figure 2 once more and address the purpose of splitting the instance into training and testing sets (X, Y). Recall we have assumed a fixed symbolic language of expressions involving Boolean combinations of some basic rectangles in the plane. This language is able to represent some regions of the plane using very succinct expressions and for representing other regions it requires less succinct expressions. For this example, let us equate *succinctness* with *syntactic length*, e.g. in Figure 2 the disjunction $r_1 \vee r_2$ is syntactically smaller, and thus more succinct, than the conjunction $r_3 \wedge r_4 \wedge r_5$.

Now, imagine we want to solve the learning instance using a fixed learning algorithm which searches the space of expressible concepts by first considering small expressions and only later considering large expressions if no small expression can be found which is consistent with a small number of examples. Many expression learning algorithms and synthesis tools use such heuristics, with syntax tree size and depth being common measures of succinctness. *For such a fixed algorithm,*

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

along with a given set of learning instances, our goal will be to synthesize a DSL such that learning succeeds *using the fixed algorithm together with the DSL*, and where *success* corresponds to finding an expression that is consistent with *all* the examples in the given learning instance, but where the learning algorithm itself only considers a smaller number of training examples. In other words, we ask for a DSL in which the most *succinct* concepts that it expresses which are consistent with the training examples X are also consistent with the testing examples Y. Learning algorithms which prefer concepts that are simple to express, when operating over such DSLs, will discover solutions which solve training examples *and* which also generalize to testing examples.

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

4.2 Base Language and Consistency

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

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

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

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

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

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

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

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

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

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

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

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

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

4.3 DSL Spaces and Properties: Adequacy and Generalization

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

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

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

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*.

Example 4.3.1 (Plain grammars and macro grammars)**.** Imagine we want a DSL that expresses specific Boolean functions like binary XOR over a fixed set of Boolean variables $x_1, \ldots, x_k$. A plain grammar cannot express such a function of some arguments. For instance, grammar rules like

$$S \;\to\; (V \wedge \neg V) \vee (\neg V \wedge V) \qquad \text{and} \qquad V \;\to\; x_1 \;\mid\; \cdots \;\mid\; x_k$$

fail to capture the requirement that the arguments to XOR are duplicated in two places in the expression which defines it, e.g. it admits $x_1 \wedge \neg x_2 \vee \neg x_3 \wedge x_4$ which is not an instance of XOR. On the other hand, a macro grammar (see Definition 3.1.2) allows nonterminals to use parameters, which permit exactly capturing XOR with rules like

$$S(1,2) \;\to\; (1 \wedge \neg 2) \vee (\neg 2 \wedge 1) \qquad \text{and} \qquad V \;\to\; x_1 \;\mid\; \cdots \;\mid\; x_k.$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[6]Many synthesis heuristics explore grammars by increasing depth.
[7]See Appendix A for an example; we omit a formal definition of parse trees.
[8]The ordering $\leq$ on numbers being the usual one extended to $\mathbb{N} \cup \{\infty\}$.

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

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

which allows all Boolean combinations of rectangles, we have that

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

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

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

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

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

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

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

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

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

4.4 Constraining the DSL Space

The problems we introduce in Sections 5 to 7 are very general. They are parameterized by a base language, which is used to define the syntax and semantics of synthesized DSLs (Section 4.2). In general, the base language should be relatively expressive, as we do not want to assume knowledge of relevant concepts in the domain, their relationships, and how much power is needed to define them. With an expressive base language, e.g. a Turing-complete programming language, there is no mechanism in the problem specification (though the algorithms we introduce can easily output the *syntactically shortest* DSL solving the problem) which prevents a synthesized DSL from "memorizing" solutions to learning instances by introducing new symbols that exactly define specific solutions, effectively making the solutions into constants and therefore as succinct as possible.

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

$$S \rightarrow \mathsf{solution}_1 \mid \cdots \mid S \rightarrow \mathsf{solution}_k.$$

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

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

Note also that handling a meta-grammar in the input is a *feature* and makes the problems more general, as it can always be omitted and an unconstrained grammar can be used by default[9].

4.5 Tree Automaton-Computable Language Semantics

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

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

Definition 4.5.1 (Tree Automaton-Computable Semantics)**.** Fix a base language consisting of grammar $G = (S, N, \Delta, P)$, examples $\mathcal{M}$, and predicate $\mathsf{consistent} \subseteq L(G) \times \mathcal{M}$. 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 $A_{\neg M}$ as *example automata*.

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

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.

[9] Appendix B shows an example of a fully permissive meta-grammar.

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 4.3.3) 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 4.3.3: they contain an expression consistent with examples $X \cup Y$, where $I = (X, Y)$. Using these A_I, we then construct a final automaton A that is the product over all A_I and also $A(\mathcal{G})$, an automaton accepting grammars that satisfy the constraint $\mathcal{G}$. This final automaton A accepts exactly the grammars satisfying $\mathcal{G}$ which contain solutions to each instance I and thus which solve the adequate DSL synthesis problem.

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

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

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

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

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

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

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

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

Mode 2. Read productions. States drawn from $\mathbf{M2} := Q_1 \cup \big(Q_1 \times \mathit{subterms}(P')\big)$, where
$$\mathit{subterms}(P') = \textstyle\bigcup_{(N_i, \alpha) \in P'} \mathit{subterms}(\alpha).$$

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

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 $(\mathbf{stay}, \langle q, t_i\rangle)$[10].

Any transition not described by the rules above has transition formula false. The full set of states and the initial states for the automaton are
$$Q := \mathbf{M1} \cup \mathbf{M2}, \qquad Q^i = \{\langle q, \mathbf{start}\rangle : q \in Q_1^i\} \subseteq \mathbf{M1}.$$

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

Proof. Follows easily by construction. See Appendix C.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 4.5.1). 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
$$A := A(\mathcal{G}) \times \mathsf{convert}\big(\textstyle\bigtimes_{p \in [l]} A_{I_p}\big),$$
where $\mathsf{convert}(B)$ is a procedure for converting a two-way alternating tree automaton B to a top-down non-deterministic automaton in time $\exp(|B|)$, as explained in [Cachat 2002; Vardi 1998]. By construction and Lemma 5.1 we have
$$L(A) = \big\{t \in L(\mathcal{G}) : \textstyle\bigwedge_{p \in [l]} \mathsf{solves}(\mathsf{extend}(\mathsf{dec}(t), G'), I_p)\big\}.$$
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. □

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

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

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

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

Proof. Follows the same logic as the construction of A_I but leaves out handling a base grammar. □

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

6 DSL Synthesis

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

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

Input:
- Instances $I_1, \ldots, I_l$
- 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 4.3.3) and generalizing (Definition 4.3.9) and
(2) $\mathsf{enc}(G) \in L(\mathcal{G})$, i.e. constraints $\mathcal{G}$ are satisfied

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

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

6.1 Equivalence of Grammars

If there is to exist an automaton that accepts exactly the grammars solving a DSL synthesis problem, then it must be possible to partition the space of grammars into finitely-many equivalence classes based on their behavior over an instance I. For *adequate* DSL synthesis the "behavior" of interest was whether or not a grammar expresses at least one solution for each learning problem.

What would make two distinct grammars G_1 and G_2 equivalent with respect to an instance $I = (X, Y)$ under the stronger requirement of generalization (Definition 4.3.9)? Whether G_1 and

G_2 are equivalent on I depends on the ease with which they express different concepts relevant to the examples in X and Y. Consider again the example from Figure 2 with $I = (P, N)$. Suppose $G_1 = (S, \{S\}, \Delta, P_1)$ and $G_2 = (S, \{S\}, \Delta, P_2)$, with $\Delta = \{\wedge^2, \vee^2, \neg^1\} \cup \{r^0 \,:\, r \in R\}$ and $R = \{r_1, r_2, r_3, r_4, r_5\}$ being a finite set of rectangles in the plane. Suppose the rules P_1 and P_2 are

$$P_1 \,:\, S \longrightarrow S \wedge S \;|\; r \in R \qquad\qquad P_2 \,:\, S \longrightarrow \neg(\neg S \vee \neg S) \;|\; r \in R.$$

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

$G_1 : S \longrightarrow S \wedge S \;|\; r \in R$

d	S
1	$\{v_{r_1}, \ldots\}$
2	$\{v_{r_1 \wedge r_2}, \ldots\}$
$\vdots$	$\vdots$

$\equiv$

$G_2 : S \longrightarrow \neg(\neg S \vee \neg S) \;|\; r \in R$

d	S
1	$\{v_{r_1}, \ldots\}$
2	$\{v_{\neg(\neg r_1 \vee \neg r_2)}, \ldots\}$
$\vdots$	$\vdots$

$\not\equiv$

$G_3 : S \longrightarrow S \wedge S \;|\; S \vee S \;|\; r \in R$

d	S
1	$\{v_{r_1}, \ldots\}$
2	$\{v_{r_1 \vee r_2}, \ldots\}$
3	$\{v_{r_3 \wedge (r_4 \wedge r_5)}, \ldots\}$
$\vdots$	$\vdots$

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

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

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

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

These tables give us a notion of equivalence that captures whether two grammars have the same expressive power, parameterized by *parse tree depth*, over fixed structures. We use the information captured by such tables, albeit with a more complex domain D, in our automaton construction for the proof of decidability.

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

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

$$T[2, S] \coloneqq T[1, S] \cup \{v \wedge v' \,:\, v, v' \in T[1, S]\} \cup \{(\mathbf{1}_{p_1 \in [\![r]\!]}, \ldots, \mathbf{1}_{p_9 \in [\![r]\!]}) \,:\, r \in R\},$$

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

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

We now describe the details of this construction further.

6.2 Automaton Construction

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

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

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

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* for I.

Recursion Tables. Let us fix a grammar $G = (S, N, \Delta, P)$. To define its recursion table $T(G)$, we order its nonterminals as $N_1, N_2, \ldots, N_k$, with $N_1 = S$. Now let $H_i : \mathcal{P}(Q_1 \sqcup Q_2)^k \longrightarrow \mathcal{P}(Q_1 \sqcup Q_2)$ 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[11] 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 $H : \mathcal{P}(Q_1 \sqcup Q_2)^k \longrightarrow \mathcal{P}(Q_1 \sqcup Q_2)^k$ defined by $H(R) = (H_1(R), \ldots, H_k(R))$ is monotone with respect to component-wise inclusion of sets, and thus the following sequence converges to a fixpoint after $n \leq k(|Q_1| + |Q_2|)$ steps:

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

We define the recursion table $T(G)$ as follows. There are $k = |N|$ columns and $n^* + 1$ rows, where $n^* := k(|Q_1| + |Q_2|)$. The entry at row i, column j, denoted $T(G)[i, j]$[12], consists of the subset of values from $Q_1 \sqcup Q_2$ that are first achieved at parse tree depth i for nonterminal N_j. For $1 \leq j \leq k$:

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

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

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

That is, the grammar G solves I if and only if there is some depth i at which it generates a solution for $X \cup Y$ and *all* non-generalizing expressions cannot be generated in depth less than i. Let us say $T(G)$ is *acceptable* if this holds.

In Appendix D we define a tree automaton A_I whose language is

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

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

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

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

The details of the construction can be found in Appendix D. We note that the number of states for A_I is exponential in the sizes of the instance automata, as the entries of the recursion table range over subsets of their states.

6.3 Decidability of DSL Synthesis

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

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

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

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

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

Lemma 6.3. *Given tree automata A and B, there is a tree automaton C that accepts an encoding of a grammar G if and only if there is some $i \in \mathbb{N}$ such that $L(A) \cap L(G)_i \neq \emptyset$ and $L(B) \cap L(G)_i = \emptyset$, where $L(G)_i$ is the set of terms obtained at iteration i of the fixpoint computation for $L(G)$.*

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

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.

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

7 DSL Synthesis for Macro Grammars

We now introduce variants of the 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 Example 4.3.1 and discussed syntax and semantics in Section 3.1. We establish decidability results for each variant.

7.1 Adequate DSL Synthesis with Macros

Problem (Adequate DSL Synthesis with Macros).
Parameters:

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

Input: Instances $I_1, \ldots, I_l$ 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 4.3.3) and
(2) $\mathsf{enc}(G) \in L(\mathcal{G})$, i.e. constraints $\mathcal{G}$ are satisfied

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

Macros lead us to a more complex decision procedure for DSL synthesis— we briefly explain the adjustments needed to prove decidability— a complete construction can be found in Appendix E.

The proof of Theorem 7.1 is similar to that of adequate DSL synthesis from Section 5, except A_I uses exponentially more states to deal with macros. To simulate the instance automaton A_1 over an input grammar and verify existence of a single expression solving I, automaton A_I keeps track of *sets* of distinct expressions generated by a given nonterminal. To see why, consider the grammar

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

Imagine A_I is reading $S \to H(G)$ and checking that S generates an expression evaluating to $q \in Q_1$, a state of A_1. It might check that an expression generated in H can evaluate to q, *assuming* that the argument G generates an expression evalutating to some $q_G \in Q_1$. Notice, however, that $H(G) \Longrightarrow h(G,G) \Longrightarrow h(a,b)$ is a valid outermost derivation, and G generated two distinct expressions despite being passed once as an argument to H. The automaton can handle this by tracking *the entire subset of* Q_1 that expressions generated by G can evaluate to. Besides this increase in states to handle macros, the construction and decision procedure are similar to that of Section 5.

7.2 DSL Synthesis with Macros

Here we define DSL synthesis with macros and state decidability for depth ordering. The argument shares much of the structure of Section 6.2 and can be found in Appendix E.2.

Problem (DSL Synthesis with Macros).
Parameters:

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

Input: Instances $I_1, \ldots, I_l$ 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 4.3.3) and generalizing (Definition 4.3.9) and
(2) $\mathsf{enc}(G) \in L(\mathcal{G})$, i.e. constraints $\mathcal{G}$ are satisfied

The new challenge in this setting is to account for an interaction between the relative succinctness constraint (Definition 4.3.9) and potentially deep nesting of macro applications. Our result below holds for classes of macro grammars where all grammar rules have macro application nesting depths bounded by a constant.

Theorem 7.2. *DSL synthesis with macros is decidable for depth ordering over any language whose semantics on fixed structures can be evaluated by tree automata (Definition 4.5.1) for any class of macro grammars whose macro nesting depth is bounded (Appendix E.2). Furthermore, the set of solutions corresponds to a regular set of trees.*

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

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

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

8 Related Work

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

Work on library learning explores the problem of compressing a given corpus of programs [Bowers et al. 2023; Cao et al. 2023] or refactoring knowledge bases expressed as logic programs [Dumancic et al. 2021], where the goal is to find a 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 a set of programs is given as input and the synthesizer must produce an *equivalent* set of programs. 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. Furthermore, we introduce a new signal related to the *relative succinctness* of concepts as expressed by a DSL (see Section 4.1), and our formulation permits a *decrease in the expressive power of the language*; this is not available to a system like DreamCoder.

Grammar induction. There is a large body of work on the synthesis of grammar/automata representations of formal languages, e.g., the L* [Angluin 1987] and RPNI [Oncina and García 1992] algorithms for learning representations of regular languages in terms of DFAs. Recent research explores applications to fuzz testing and learning program input grammars [Bastani et al. 2017; Kulkarni et al. 2021; Miltner et al. 2023]. Vanlehn and Ball [Vanlehn and Ball 1987] explored an approach to context-free grammar induction based on version space algebra [Mitchell 1982]. The tree automata in our decidability proofs can be understood as version spaces, one per input learning instance, where each instance-specific automaton represents the set of all grammars (i.e. DSLs) which solve and generalize on that learning instance.

Though we also consider learning grammars from data, our problems are very different from grammar induction, where the specification is driven by the syntax of examples, i.e., character sequences. In contrast, for our problems the relevant properties of grammars are driven by language *semantics* and its relationship with input examples. Most importantly, we look for grammars with specific biases specified by few-shot learning problems split into training and testing sets.

Applications of tree automata to synthesis. 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]. The tree automata in this work read parse trees of expressions in order to check whether they satisfy a fixed example, and there is no restriction of the examples to efficient classes of structures, in contrast to applications of tree automata in model checking over parameterized classes of structures [Courcelle and Engelfriet 2012]. 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], and as an algorithmic framework in program synthesis tools, e.g., [Gulwani 2011; Polozov and Gulwani 2015; Wang et al. 2017b, 2018].

9 Conclusion

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

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

In future work we plan to explore practical implementations for DSL synthesis. It will be interesting to test whether 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.

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

Adam Betts, Nathan Chong, Alastair Donaldson, Shaz Qadeer, and Paul Thomson. 2012. GPUVerify: a verifier for GPU kernels. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)*. Association for Computing Machinery, New York, NY, USA, 113–132. https://doi.org/10.1145/2384616.2384625

James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. 2016. Specifying and Checking File System Crash-Consistency Models. (2016), 83–98. https://doi.org/10.1145/2872362.2872406

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

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

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

José Cambronero, Sumit Gulwani, Vu Le, Daniel Perelman, Arjun Radhakrishna, Clint Simon, and Ashish Tiwari. 2023. FlashFill++: Scaling Programming by Example by Cutting to the Chase. *Proc. ACM Program. Lang.* 7, POPL, Article 33 (Jan. 2023), 30 pages. https://doi.org/10.1145/3571226

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

Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. 2016. Programmatic and direct manipulation, together at last. *SIGPLAN Not.* 51, 6 (June 2016), 341–354. https://doi.org/10.1145/2980983.2908103

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

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

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

Cormac Flanagan and K. Rustan M. Leino. 2001. Houdini, an Annotation Assistant for ESC/Java. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity (FME '01)*. Springer-Verlag, Berlin, Heidelberg, 500–517.

Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. 2014. ICE: A Robust Framework for Learning Invariants. In *Computer Aided Verification*, Armin Biere and Roderick Bloem (Eds.). Springer International Publishing, Cham, 69–87.

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. Association for Computing Machinery, New York, NY, USA, 317–330. https://doi.org/10.1145/1926385.1926423

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

Vu Le and Sumit Gulwani. 2014. FlashExtract: a framework for data extraction by examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 542–553. https://doi.org/10.1145/2594291.2594333

Sirui Lu and Rastislav Bodík. 2023. Grisette: Symbolic Compilation as a Functional Programming Library. *Proc. ACM Program. Lang.* 7, POPL, Article 16 (Jan. 2023), 33 pages. https://doi.org/10.1145/3571209

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

Anders Miltner, Devon Loehr, Arnold Mong, Kathleen Fisher, and David Walker. 2023. 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.

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

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

Emina Torlak and Rastislav Bodik. 2013. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2013)*. Association for Computing Machinery, New York, NY, USA, 135–152. https://doi.org/10.1145/2509578.2509586

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

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

Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017a. Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*

(PLDI 2017). Association for Computing Machinery, New York, NY, USA, 452–466. https://doi.org/10.1145/3062341.3062365

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

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

He Zhu, Stephen Magill, and Suresh Jagannathan. 2018. A data-driven CHC solver. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 707–721. https://doi.org/10.1145/3192366.3192416

A Section 3

A.1 Encoding grammars as 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.

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

A.2 Decoding trees into grammars

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.

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

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.

B Section 4

B.1 Example meta-grammar

Figure 4 shows and example of a fully permissive meta-grammar over alphabet $\Gamma(\Delta, N)$.

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

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

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':

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

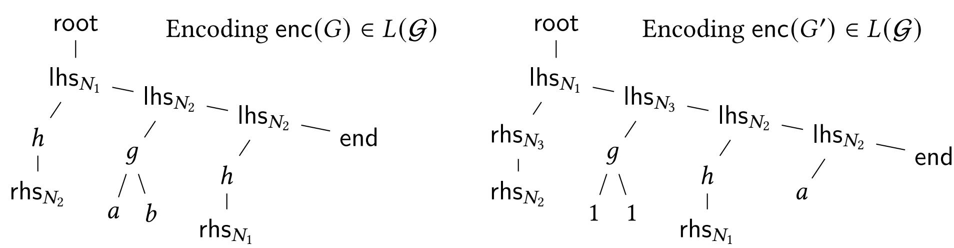

Fig. 4. (Top right) Tree grammars G and G' over $\Delta = \{a^0, b^0, h^1, g^2\}$ and nonterminals $N = \{N_1^0, N_2^0, N_3^1\}$ and (bottom) their encodings as trees $\mathsf{enc}(G)$ and $\mathsf{enc}(G')$ over alphabet $\Gamma(\Delta, N)$. (Top left) A meta-grammar $\mathcal{G}$ over alphabet $\Gamma(\Delta, N)$ with $\mathsf{enc}(G) \in L(\mathcal{G})$.

B.2 Examples of parse trees

We omit a (trivial) formalization of *parse trees* for terms and tree macro grammars. Examples of parse trees for $h(h(h(a)))$ and $f(f(a))$ in the following macro grammar are shown in Figure 5. Note that for determining parse tree depth with respect to macro grammars, we do not consider branches of the parse tree which record the arguments to a macro (the red edge in Figure 5). This is because macro definitions may discard any of their arguments, and in such cases the depth of the parse trees for the arguments should not count toward the depth of the parse tree for a macro application because we are considering outermost derivations.

$$
\begin{aligned}
S &\longrightarrow H(A) \mid f(f(S)) \mid A \\
H(1) &\longrightarrow h(h(h(1))) \\
A &\longrightarrow a
\end{aligned}
$$

C Section 5

C.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 $A_1 = (Q_1, \Delta, I_1, \delta_1)$.

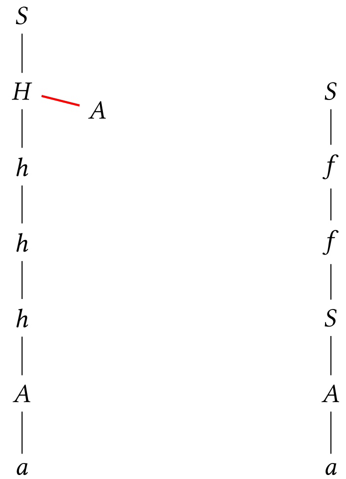

Fig. 5. Parse trees for terms $h(h(h(a)))$ and $f(f(a))$. Arguments for macros are hanging to the right.

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.

D Section 6

We define a 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.

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.

D.1 Construction of A_I

We now precisely define the 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. We assume k nonterminal symbols. 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:

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

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

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

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

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

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

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

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

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

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

Transitions for **M4a**:

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

- $\delta(\langle u, \langle L, m\rangle\rangle, \mathsf{rhs}_{N_i}) = \mathsf{true} \quad \text{if } u \in L_i$
- $\delta(\langle u, \langle L, m\rangle\rangle, \mathsf{rhs}_{N_i}) = \mathsf{false} \quad \text{if } u \notin L_i$

Transitions for **M4b**:

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

The notation $\mathit{adorn}(s, \varphi)$ represents the transition formula obtained by replacing each atom (i, q) in the Boolean formula φ by the atom $(i, \langle q, s\rangle)$. The notation $\mathit{adorn}'(\langle s, t_1, \ldots, t_r\rangle, \varphi)$ represents the transition formula obtained by replacing each atom (i, q) in φ by the atom $(\mathbf{stay}, \langle q, \langle s, t_i\rangle\rangle)$[13].

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

$$\mathbf{M5} := \mathbf{M5a} \cup \mathbf{M5b}$$

$$\mathbf{M5a} := D^k \times D \times \{\mathbf{miss}\} \cup ((Q_1 \sqcup Q_2) \times (D^k \times \{1, 2\} \times \{\bot\}))$$

$$\mathbf{M5b} := \big(D^k \times D \times \mathit{subterms}(P') \times \{\mathbf{miss}\}\big) \cup \big((Q_1 \sqcup Q_2) \times (D^k \times \{1, 2\} \times \{\bot\} \times \mathit{subterms}(P'))\big)$$

Transitions for **M5a**:

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

Transitions for **M5b**:

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

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

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

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

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

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

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

By construction, we have the following.

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

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

Proof. Appendix D.2. □

D.2 Proof of Lemma D.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.

E Section 7

E.1 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 outermost derivations, be arbitrary terms involving other macros, our automaton will keep track of sets of states which the parameters may evaluate to.

When evaluating a macro application, the automaton verifies that *sets* of states $Q' \subseteq Q_1$ are achievable, with Q_1 the states for A_1. For example, when reading a macro $F(t_1, t_2)$, A_X 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.

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 $t_1, \dots, t_r \in T_\Delta(\{\mathsf{rhs}_{N_i} : N_i \in N^{=0}\})$. 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

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

$$\begin{aligned}
\delta(\langle C, \mathbf{start}\rangle, \mathsf{root}) &= (\mathbf{down}, \langle C, \mathbf{start}\rangle)\\
\delta(\langle C, \mathbf{start}\rangle, \mathsf{lhs}_{N_i}) &= (\mathbf{stay}, \langle C, N_i\rangle)\\
\delta(\langle C, N_i\rangle, \mathsf{lhs}_{Y}) &= \vee_{\{(C_1,C_2):C_1\cup C_2=C\}} (\mathbf{up}, \langle C_1, N_i\rangle) \vee (\mathbf{right}, \langle C_2, N_i\rangle)\\
\delta(\langle C, N_i\rangle, \mathsf{lhs}_{N_i}) &= \vee_{\{(C_1,C_2,C_3,C_4):C_1\cup C_2\cup C_3\cup C_4=C\}} (\mathbf{up}, \langle C_1, N_i\rangle) \wedge (\mathbf{left}, C_2) \wedge (\mathbf{right}, \langle C_3, N_i\rangle)\\
&\qquad \wedge_{q\in C_4} \big(\vee_{(N_i,\alpha)\in P'} (\mathbf{stay}, \langle \{q\}, \alpha\rangle)\big)\\
\delta(\langle C, N_i\rangle, _) &= \vee_{\{(C_1,C_2):C_1\cup C_2=C\}} (\mathbf{up}, \langle C_1, N_i\rangle) \wedge_{q\in C_2} \big(\vee_{(N_i,\alpha)\in P'} (\mathbf{stay}, \langle \{q\}, \alpha\rangle)\big)\\
\delta(\langle C_1, \dots, C_k, C, F\rangle, \mathsf{lhs}_{Y'}) &= \vee_{\{(C'_1,C'_2):C'_1\cup C'_2=C\}} (\mathbf{up}, \langle C_1, \dots, C_k, C'_1, F\rangle) \wedge (\mathbf{right}, \langle C_1, \dots, C_k, C'_2, F\rangle)\\
\delta(\langle C_1, \dots, C_k, C, F\rangle, \mathsf{lhs}_{F}) &= \vee_{\{(C'_1,C'_2,C'_3):C'_1\cup C'_2\cup C'_3=C\}}\\
&\qquad (\mathbf{up}, \langle C_1, \dots, C_k, C'_1, F\rangle)\\
&\qquad \wedge (\mathbf{left}, \langle C_1, \dots, C_k, C'_2, F\rangle) \wedge (\mathbf{right}, \langle C_1, \dots, C_k, C'_3, F\rangle)\\
\delta(\langle C_1, \dots, C_k, C, F\rangle, _) &= (\mathbf{up}, \langle C_1, \dots, C_k, C, F\rangle) \text{ // base language is macro-less}
\end{aligned}$$

Mode 2. Read productions. States drawn from

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

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

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

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

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

We have the following by construction.

Lemma E.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 E.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 E.3. *For the languages covered in Theorem E.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.2 DSL synthesis with macros

The *macro depth* of a tree $t \in T_{\Delta \cup N}(\mathbb{N})$ is the maximum number of macro symbols encountered along any root-to-leaf path in t. For example, $f(a)$, $N(a)$, and $N(1, H(2))$ have macro depths 0, 1, and 2, respectively. A macro grammar *has macro depth bounded by b* if, for all of its rules (N, α), α has macro depth no more than b. Note that given a macro grammar G and bound $b \in \mathbb{N}$, we can easily verify that G has macro depth bounded by b.

A meta-grammar $\mathcal{G}$ has macro depth bounded by $b \in \mathbb{N}$ if, for every $G \in L(\mathcal{G})$, G has macro depth bounded by b. Given a meta-grammar $\mathcal{G}$, we can compute the minimum b (if it exists) for which it has macro depth bounded by b. We can do it by checking for "cycles" in the rules corresponding to right-hand sides of object grammar rules which contain macro symbols along them. We can keep track of the maximum macro depth possible while looking for cycles. If such a cycle is found then there is no bound b and if not we output the maximum macro depth as the bound b.

The proof of Theorem 7.2 is 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 $I = (X, Y)$ for parse tree depth ordering. Because macro grammars allow applications of macros to be nested, and because this problem requires paying attention to the depth of expressions, the automaton A_I uses many more states than the construction from Section 6.3 in order to handle nested macros. We describe a construction of A_I, which is parameterized by a depth bound $b \in \mathbb{N}$, such that A_I operates as expected over grammars with macro nesting depth bounded by b.

The construction shares much of the structure related to recursion tables, but in building up each row one after the other, the automaton must keep fine-grained information about a bounded number of previous rows to handle boundedly-nested macros. Additionally, the entries of the recursion table corresponding to macro symbols indicate *functions* from tuples of value sets 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 several previous rows of the recursion table up to each individual depth. This is sufficient to implement the same protocol for nondeterministically guessing and verifying the rows of the recursion table for a macro grammar.

E.2.1 Construction of A_I . As before we assume non-deterministic top-down instance automata $A_1 = (Q_1, \Delta, Q_1^i, \delta_1)$ and $A_2 = (Q_2, \Delta, Q_2^i, \delta_2)$. The automaton A_I will guess the construction of tables similar to the recursion tables of Section 6.3. A difference is that the entries for *non-macro nonterminals* range over the powerset $\mathcal{P}(Q_1 \sqcup Q_2)$ and the entries for *macro nonterminals* range over functions over these sets. Let us use $D \coloneqq \mathcal{P}(Q_1 \sqcup Q_2)$ as shorthand in the remainder. Fix a macro nesting depth bound $b \in \mathbb{N}$. Consider a macro symbol F^1. Entries for its column in the table range over functions $[D \longrightarrow D]$. For a symbol K^2 they range over $[D^2 \longrightarrow D]$, etc. For any macro grammar G, the intuition is that if $T(G)[i, j] = f \in [D \longrightarrow D]$, then, provided an argument which can generate values $v \in D$, the nonterminal N_j can generate terms evaluating to values $f(v)$ using derivations of depth i and no smaller.

We now define the 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. We describe the states Q and their transition formulas grouped by functionality. The transitions are similar to those of Section 6.3 and are organized similarly as well. We note salient differences alongside the transitions. We assume there are k nonterminal symbols. 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.

We assume a macro nesting depth bound of $b \in \mathbb{N}$. We assume table rows are drawn from a set *Rows*. We assume that the entries of rows which correspond to macro symbols contain vectors representing the values the functions take for some fixed ordering of the elements of their finite input sets. For instance, for a binary macro symbol $G(1, 2)$, its entries in the table correspond to functions of type $D \times D \longrightarrow D$, but are manipulated in the construction as vectors $v \in D^l$, where $l = |D \times D|$. We write *ArgDomain* to mean the domain of vectors of arguments for macro symbols, e.g. it includes $D \times D$ for binary macro symbols, etc. We write *dropFirst* to mean dropping the earliest (first) row in a sequence of rows. We write *emptyRow* to mean the row whose entries are all empty; for a vector corresponding to a function this means a vector of empty sets. We write $\emptyset$ to mean either an empty set or a vector of empty sets, depending on the context. We write *fullRow* to mean the row whose entries are all full; for sets this is $Q_1 \sqcup Q_2$ and for vectors over such sets which model functions this is $(Q_1 \sqcup Q_2)^l$, for an appropriate vector length l depending on the number of parameters for the macro symbol at a particular index in the row. In some cases we use $\cup$ to mean both union of sets and componentwise union for vectors over sets, depending on context (e.g. first bullet of Appendix E.2.1).

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

$$\mathbf{M1} := \mathit{Rows}^{b+3} \times \{\mathbf{reset}, \mathbf{start}\}.$$

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

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

$$\mathbf{M2} := \mathit{Rows}^{b+3} \times [k] \times \{\mathbf{row}\}.$$

// Forget the deepest row (*dropFirst*) of the table as we add a new one
- $\delta(\langle \mathit{prev}, L, C, R, i, \mathbf{row}\rangle, \_) = \bigvee_{(C',R') \in \mathbf{split}(R)} (\mathbf{stay}, \langle \mathit{dropFirst}(\mathit{prev}), L, L \cup C, C', C', R', i, \mathbf{prod}\rangle)$
with $\mathbf{split}(R) := \big\{(C', R') \in \mathit{Rows} \times \mathit{Rows} \;:\; C' \cup R' = R,\ C' \neq \mathit{emptyRow}\big\}$

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

$$\mathbf{M3} := \mathit{Rows}^{b+4} \times [k] \times \{\mathbf{prod}\}.$$

- $\delta(\langle \mathit{prev}, L, C, W, R, i, \mathbf{prod}\rangle, \mathsf{lhs}_{N_j^0}) = \bigvee_{\{(U,V)\,:\,U \cup V = C_j\}} (\mathbf{left}, \langle \mathit{prev}, L, \mathit{noArgs}, U, \mathbf{hit}\rangle) \wedge$
$(\mathbf{left}, \langle \mathit{prev}, L, R_j, \mathbf{miss}\rangle) \wedge$
$(\mathbf{right}, \langle \mathit{prev}, L, \langle C_1, \ldots, C_{j-1}, V, \ldots, C_k\rangle, W, R, i, \mathbf{prod}\rangle)$
- // Compute the macro arguments using $\mathit{args}(j, \mathit{idx})$ to give to the child state
$\delta(\langle \mathit{prev}, L, C, W, R, i, \mathbf{prod}\rangle, \mathsf{lhs}_{N_j^{>0}}) = \bigvee_{\{(U,V)\,:\,U \cup V = C_j\}}$
$\Big(\bigwedge_{\mathit{idx} \in [|C_j|]} (\mathbf{left}, \langle \mathit{prev}, L, \mathit{args}(j, \mathit{idx}), U, \mathbf{hit}\rangle)\Big) \wedge (\mathbf{left}, \langle \mathit{prev}, L, R_j, \mathbf{miss}\rangle) \wedge$
$(\mathbf{right}, \langle \mathit{prev}, L, \langle C_1, \ldots, C_{j-1}, V, \ldots, C_k\rangle, W, R, i, \mathbf{prod}\rangle)$
- $\delta(\langle \mathit{prev}, L, C, W, R, i, \mathbf{prod}\rangle, \mathsf{end}) =$
if $\exists (N_j \in N \setminus N').\, C_j \neq \emptyset$
then false
else // Note: i indexes the starting nonterminal

$$\Big((\mathbf{stay}, \langle L_i \cup W_i, \mathbf{solve}\rangle) \vee \big((\mathbf{stay}, \langle \mathit{prev}, L, W, R, \mathbf{reset}\rangle) \wedge (\mathbf{stay}, \langle L_i \cup W_i, \mathbf{ok}\rangle)\big)\Big)$$
$$\wedge \Big(\bigwedge_{N_j \in (N')^0} \bigwedge_{u \in C_j} \bigvee_{(N_j,\alpha) \in P'} (\mathbf{stay}, \langle \mathit{prev}, L, \mathit{noArgs}, \{u\}, \alpha, \mathbf{hit}\rangle)\Big)$$
$$\wedge \Big(\bigwedge_{N_j \in (N')^{>0}} \bigwedge_{\mathit{idx} \in [|C_j|]} \bigwedge_{u \in C_j(\mathit{idx})} \bigvee_{(N_j,\alpha) \in P'} (\mathbf{stay}, \langle \mathit{prev}, L, \mathit{args}(j, \mathit{idx}), \{u\}, \alpha, \mathbf{hit}\rangle)\Big)$$
$$\wedge \Big(\bigwedge_{N_j \in N'} \bigwedge_{(N_j,\alpha) \in P'} (\mathbf{stay}, \langle \mathit{prev}, L, R_j, \alpha, \mathbf{miss}\rangle)\Big)$$

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

$$\begin{aligned}
\mathbf{M4} &:= \mathbf{M4a} \cup \mathbf{M4b} \\
\mathbf{M4a} &:= \mathit{Rows}^b \times \mathit{Rows} \times \mathit{ArgDomain} \times D \times \{\mathbf{hit}\} \\
&\quad \cup ((Q_1 \sqcup Q_2) \times (\mathit{Rows}^{b+1} \times \mathit{ArgDomain} \times \{0, \ldots, b-1\} \times \{1, 2\})) \\
\mathbf{M4b} &:= \big(\mathit{Rows}^b \times \mathit{Rows} \times \mathit{ArgDomain} \times D \times \mathit{subterms}(P') \times \{\mathbf{hit}\}\big) \\
&\quad \cup \big((Q_1 \sqcup Q_2) \times (\mathit{Rows}^{b+1} \times \mathit{ArgDomain} \times \{0, \ldots, b-1\} \times \{1, 2\} \times \mathit{subterms}(P'))\big), \\
&\quad \text{where } \mathit{subterms}(P') = \textstyle\bigcup_{(N_i, \alpha) \in P'} \mathit{subterms}(\alpha)
\end{aligned}$$

Transitions for **M4a**:

- // Current macro nesting depth starts at 0
$\delta(\langle \mathit{prev}, L, \mathit{topArgs}, U, \mathbf{hit}\rangle, \_) = \bigwedge_{u_1 \in U \cap Q_1} (\mathbf{stay}, \langle u_1, \langle \mathit{prev}, L, \mathit{topArgs}, \mathit{currDepth} = 0, 1\rangle\rangle) \wedge \bigwedge_{u_2 \in U \cap Q_2} (\mathbf{stay}, \langle u_2, \langle \mathit{prev}, L, \mathit{topArgs}, \mathit{currDepth} = 0, 2\rangle\rangle)$
- $\delta(\langle u, \langle \mathit{prev}, L, \mathit{topArgs}, \mathit{currDepth}, m\rangle\rangle, x) =$
$\mathit{adorn}(\langle \mathit{prev}, L, \mathit{topArgs}, \mathit{currDepth}, m\rangle, \delta_m(u, x)), \quad x \in \Delta$
- // Guess how much available depth to budget for arguments and how much for macro expansion
$\delta(\langle u, \langle \mathit{prev}, L, \mathit{topArgs}, \mathit{currDepth}, m\rangle\rangle, \mathsf{rhs}_{N_i^{>0}}) = \bigvee_{\mathit{offset} \in [b - \mathit{currDepth}]} \bigvee_{\{\mathit{idx} \in [|\mathit{target}|] : u \in \mathit{target}(\mathit{idx})\}} \bigwedge_{p \in [|\mathit{args}(i, \mathit{idx})|]} \bigwedge_{u' \in \mathit{args}(i, \mathit{idx})(p)} (p, \langle u', \langle \mathit{prev}, L, \mathit{topArgs}, \mathit{currDepth} + \mathit{offset}, m\rangle\rangle)$
// Do not need to check the arguments evaluate precisely to chosen ones
where $\mathit{target} = \langle \mathit{prev}, L\rangle(b - \mathit{currDepth})(i)$
- $\delta(\langle u, \langle \mathit{prev}, L, \mathit{topArgs}, \mathit{currDepth}, m\rangle\rangle, \mathsf{rhs}_{N_i^0}) = \mathsf{true} \quad \text{if } u \in \mathit{target}$
$\delta(\langle u, \langle \mathit{prev}, L, \mathit{topArgs}, \mathit{currDepth}, m\rangle\rangle, \mathsf{rhs}_{N_i^0}) = \mathsf{false} \quad \text{if } u \notin \mathit{target}$
where $\mathit{target} = \langle \mathit{prev}, L\rangle(b - \mathit{currDepth})(i)$
- $\delta(\langle u, \langle \mathit{prev}, L, \mathit{topArgs}, \mathit{currDepth}, m\rangle\rangle, i) = \mathsf{true} \quad \text{if } u \in \mathit{topArgs}(i)$
$\delta(\langle u, \langle \mathit{prev}, L, \mathit{topArgs}, \mathit{currDepth}, m\rangle\rangle, i) = \mathsf{false} \quad \text{if } u \notin \mathit{topArgs}(i)$

  // i is a formal macro parameter

Transitions for **M4b**:

- $\delta(\langle \mathit{prev}, L, \mathit{topArgs}, U, \alpha, \mathbf{hit}\rangle, \_) = \bigwedge_{u_1 \in U \cap Q_1} (\mathbf{stay}, \langle u_1, \langle \mathit{prev}, L, \mathit{topArgs}, \mathit{currDepth} = 0, 1, \alpha\rangle\rangle) \wedge \bigwedge_{u_2 \in U \cap Q_2} (\mathbf{stay}, \langle u_2, \langle \mathit{prev}, L, \mathit{topArgs}, \mathit{currDepth} = 0, 2, \alpha\rangle\rangle)$
- $\delta(\langle u, \langle \mathit{prev}, L, \mathit{topArgs}, \mathit{currDepth}, m, f(t_1, \ldots, t_r)\rangle\rangle, \_) =$
$\mathit{adorn}'(\langle\langle \mathit{prev}, L, \mathit{topArgs}, \mathit{currDepth}, m\rangle, t_1, \ldots, t_r\rangle, \delta_m(u, f))$
- $\delta(\langle u, \langle \mathit{prev}, L, \mathit{topArgs}, \mathit{currDepth}, m, \mathsf{rhs}_{N_i^{>0}}\rangle\rangle, \_) = \bigvee_{\mathit{offset} \in [b - \mathit{currDepth}]} \bigvee_{\{\mathit{idx} \in [|\mathit{target}|] : u \in \mathit{target}(\mathit{idx})\}} \bigwedge_{p \in [|\mathit{args}(i, \mathit{idx})|]} \bigwedge_{u' \in \mathit{args}(i, \mathit{idx})(p)} (p, \langle u', \langle \mathit{prev}, L, \mathit{topArgs}, \mathit{currDepth} + \mathit{offset}, m\rangle\rangle)$
// Do not need to check the arguments evaluate precisely to chosen ones
where $\mathit{target} = \langle \mathit{prev}, L\rangle(b - \mathit{currDepth})(i)$
- $\delta(\langle u, \langle \mathit{prev}, L, \mathit{topArgs}, \mathit{currDepth}, m, \mathsf{rhs}_{N_i^0}\rangle\rangle, \_) = \mathsf{true} \quad \text{if } u \in \mathit{target}$
$\delta(\langle u, \langle \mathit{prev}, L, \mathit{topArgs}, \mathit{currDepth}, m, \mathsf{rhs}_{N_i^0}\rangle\rangle, \_) = \mathsf{false} \quad \text{if } u \notin \mathit{target}$
where $\mathit{target} = \langle \mathit{prev}, L\rangle(b - \mathit{currDepth})(i)$
- $\delta(\langle u, \langle \mathit{prev}, L, \mathit{topArgs}, \mathit{currDepth}, m, i\rangle\rangle, \_) = \mathsf{true} \quad \text{if } u \in \mathit{topArgs}(i)$
$\delta(\langle u, \langle \mathit{prev}, L, \mathit{topArgs}, \mathit{currDepth}, m, i\rangle\rangle, \_) = \mathsf{false} \quad \text{if } u \notin \mathit{topArgs}(i)$

  // i is a formal macro parameter

The notation $adorn(s, \varphi)$ 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 $adorn'(\langle s, t_1, \dots, t_r\rangle, \varphi)$ represents the transition formula obtained by replacing each atom (i, q) in φ by the atom $(\mathbf{stay}, \langle q, \langle s, t_i\rangle\rangle)$[14].

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

$$
\begin{aligned}
\mathbf{M5} &:= \mathbf{M5a} \cup \mathbf{M5b}\\
\mathbf{M5a} &:= Rows^b \times Rows \times ArgDomain \times D \times \{\mathbf{miss}\}\\
&\quad \cup ((Q_1 \sqcup Q_2) \times (Rows^{b+1} \times ArgDomain \times \{0, \dots, b-1\} \times \{1, 2\} \times \{\bot\}))\\
\mathbf{M5b} &:= \big(Rows^b \times Rows \times ArgDomain \times D \times subterms(P') \times \{\mathbf{miss}\}\big)\\
&\quad \cup \big((Q_1 \sqcup Q_2) \times (Rows^{b+1} \times ArgDomain \times \{0, \dots, b-1\} \times \{1, 2\} \times \{\bot\} \times subterms(P'))\big),\\
&\quad \text{where } subterms(P') = \textstyle\bigcup_{(N_i,\alpha)\,\in P'} subterms(\alpha)
\end{aligned}
$$

Transitions for **M5a**:

- $\delta(\langle prev, L, topArgs, U, \mathbf{miss}\rangle, \_) = \bigwedge_{u_1 \in U \cap Q_1}(\mathbf{stay}, \langle u_1, \langle prev, L, topArgs, currDepth = 0, 1, \bot\rangle\rangle) \wedge \bigwedge_{u_2 \in U \cap Q_2}(\mathbf{stay}, \langle u_2, \langle prev, L, topArgs, currDepth = 0, 2, \bot\rangle\rangle)$
- $\delta(\langle u, \langle prev, L, topArgs, currDepth, m, \bot\rangle\rangle, x) =$
  $adorn(\langle prev, L, topArgs, currDepth, m, \bot\rangle, \delta_m(u, x)), \quad x \in \Delta$
- $\delta(\langle u, \langle prev, L, topArgs, currDepth, m, \bot\rangle\rangle, \mathsf{rhs}_{N_i^{>0}}) = \bigwedge_{offset \in [b-currDepth]} \bigwedge_{\{idx \in [|target|]\,:\, u \in target(idx)\}} \bigvee_{p \in [|args(i,idx)|]} \bigvee_{u' \in args(i,idx)(p)} (p, \langle u', \langle prev, L, topArgs, currDepth + offset, m, \bot\rangle\rangle)$
  where $target = \langle prev, L\rangle(b - currDepth)(i)$
- $\delta(\langle u, \langle prev, L, topArgs, currDepth, m, \bot\rangle\rangle, \mathsf{rhs}_{N_i^0}) = \mathsf{false} \quad \text{if } u \in target$
  $\delta(\langle u, \langle prev, L, topArgs, currDepth, m, \bot\rangle\rangle, \mathsf{rhs}_{N_i^0}) = \mathsf{true} \quad \text{if } u \notin target$
  where $target = \langle prev, L\rangle(b - currDepth)(i)$
- $\delta(\langle u, \langle prev, L, topArgs, currDepth, m, \bot\rangle\rangle, i) = \mathsf{false} \quad \text{if } u \in topArgs(i)$
  $\delta(\langle u, \langle prev, L, topArgs, currDepth, m, \bot\rangle\rangle, i) = \mathsf{true} \quad \text{if } u \notin topArgs(i)$

  // i is a formal macro parameter

Transitions for **M5b**:

- $\delta(\langle prev, L, topArgs, U, \alpha, \mathbf{miss}\rangle, \_) = \bigwedge_{u_1 \in U \cap Q_1}(\mathbf{stay}, \langle u_1, \langle prev, L, topArgs, currDepth = 0, 1, \bot, \alpha\rangle\rangle) \wedge \bigwedge_{u_2 \in U \cap Q_2}(\mathbf{stay}, \langle u_2, \langle prev, L, topArgs, currDepth = 0, 2, \bot, \alpha\rangle\rangle)$
- $\delta(\langle u, \langle prev, L, topArgs, currDepth, m, \bot, f(t_1, \dots, t_r)\rangle\rangle, \_) =$
  $adorn'(\langle\langle prev, L, topArgs, currDepth, m, \bot\rangle, t_1, \dots, t_r\rangle, \delta_m(u, f))$
- $\delta(\langle u, \langle prev, L, topArgs, currDepth, m, \bot, \mathsf{rhs}_{N_i^{>0}}\rangle\rangle, \_) =$
  $\bigwedge_{offset \in [b-currDepth]} \bigwedge_{\{idx \in [|target|]\,:\, u \in target(idx)\}}$
  $\bigvee_{p \in [|args(i,idx)|]} \bigvee_{u' \in args(i,idx)(p)} (p, \langle u', \langle prev, L, topArgs, currDepth + offset, m, \bot\rangle\rangle)$
  where $target = \langle prev, L\rangle(b - currDepth)(i)$
- $\delta(\langle u, \langle prev, L, topArgs, currDepth, m, \bot, \mathsf{rhs}_{N_i^0}\rangle\rangle, \_) = \mathsf{false} \quad \text{if } u \in target$
  $\delta(\langle u, \langle prev, L, topArgs, currDepth, m, \bot, \mathsf{rhs}_{N_i^0}\rangle\rangle, \_) = \mathsf{true} \quad \text{if } u \notin target$
  where $target = \langle prev, L\rangle(b - currDepth)(i)$

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

- $\delta(\langle u, \langle \mathit{prev}, L, \mathit{topArgs}, \mathit{currDepth}, m, \bot, i\rangle\rangle, \_) = \mathsf{false} \quad \text{if } u \in \mathit{topArgs}(i)$
  $\delta(\langle u, \langle \mathit{prev}, L, \mathit{topArgs}, \mathit{currDepth}, m, \bot, i\rangle\rangle, \_) = \mathsf{true} \quad \text{if } u \notin \mathit{topArgs}(i)$

  // i is a formal macro parameter

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

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

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

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

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

$$Q \coloneqq \mathbf{M1} \sqcup \mathbf{M2} \sqcup \mathbf{M3} \sqcup \mathbf{M4} \sqcup \mathbf{M5} \sqcup \mathbf{M6} \qquad \text{and} \qquad Q_i \coloneqq \{\langle \{\mathit{emptyRow}\}^{b+2}, \mathit{fullRow}, \mathbf{reset}\rangle\}.$$

By construction we have the following.

Lemma E.4. $L(A_I) = \{t \in T_{\Gamma(\Delta,N)} \ : \ \mathsf{solves}(\mathsf{extend}(\mathsf{dec}(t), G'), I)\}$.

Proof. Similar argument to Appendix D.2. □

The rest of the proof involves standard constructions essentially identical to those of Section 6.3.