Learning Formulas in Finite Variable Logics

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

10

We consider grammar-restricted exact learning of formulas and terms in finite variable logics. We propose a novel and versatile automata-theoretic technique for solving such problems. We first show results for learning formulas that classify a set of positively- and negatively-labeled structures. We give algorithms for realizability and synthesis of such formulas along with upper and lower bounds. We also establish positive results using our technique for other logics and variants of the learning problem, including first-order logic with least fixed point definitions, higher-order logics, and synthesis of queries and terms with recursively-defined functions.

CCS Concepts: • **Theory of computation → Tree languages**; • **Computing methodologies → Machine learning approaches**.

Additional Key Words and Phrases: exact learning, learning formulas, tree automata, version space algebra, program synthesis, interpretable learning

ACM Reference Format:
Paul Krogmeier and P. Madhusudan. 2022. Learning Formulas in Finite Variable Logics. *Proc. ACM Program. Lang.* 6, POPL, Article 10 (January 2022), 28 pages. https://doi.org/10.1145/3498671

1 INTRODUCTION

Learning symbolically representable concepts from data is an important emerging area of research. Symbolic expressions, such as logical formulas or programs, can be easily analyzed and interpreted, which aids downstream applications (e.g., analyzing a large system that has a classifier as a component) and makes them easier to communicate to both humans and computers.

In this paper, we embark on a foundational study of *exact* learning of logical formulas. For a logic $\mathcal{L}$, we study the *separability* problem: given a set of positively- and negatively-labeled finite structures, we want to learn a sentence φ in $\mathcal{L}$ that is true on the positive structures and false on the negative structures. Separability consists of two related problems. First, the *realizability problem*: a decision problem that asks whether such a sentence exists, and second, the *synthesis problem*, which asks to construct a sentence if one exists. For logics that contain *infinitely many* semantically inequivalent formulas, including most of the logics considered in this paper, the realizability problem itself is not trivial.

In a learning context, one is often interested in how well a learned artifact generalizes to unseen inputs. In practice, most learning algorithms typically attempt only to minimize loss in accuracy on a set of training samples [Mitchell 1997]. Exact learning asks for a perfect classifier with respect to the training samples. Two common strategies to mitigate *overfitting* are (1) to only consider classifiers from a restricted hypothesis class $\mathcal{H}$ and (2) to prefer *simple* concepts over complex ones. The problems we study here reflect (1) by considering exact learning with respect to *grammars*. Problem

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

2475-1421/2022/1-ART10
https://doi.org/10.1145/3498671

instances are equipped with a grammar G that defines a subset $L(G)$ of logical expressions in $\mathcal{L}$ to which classifiers must belong. The problems reflect (2) by requiring a synthesizer to construct small (perhaps the smallest) formulas that separate sample structures.

We describe a very general technique for solving the realizability and synthesis problems for *several* logics with finitely many variables. In particular, a main contribution of our work is to solve realizability and synthesis for $\mathsf{FO}(k)$, a version of first-order logic with k variables. This logic allows for an arbitrary number and nesting depth of quantifiers. Although the number of variables is bounded, it is possible to *reuse* variables. For instance, consider finite graphs. Given two constants s and t and any $n \in \mathbb{N}$, the property that *t is reachable from s using at most n edges* is expressible using just two variables, and thus $\mathsf{FO}(k)$ with $k = 2$ contains an infinite set of inequivalent formulas.

We prove that for every $k \in \mathbb{N}$, the realizability problem for $\mathsf{FO}(k)$ over a grammar is decidable. That is, given a (tree) grammar G defining a subclass of $\mathsf{FO}(k)$ and sets of positively- and negatively-labeled structures *Pos* and *Neg*, it is decidable to check whether there is a sentence $\varphi \in L(G)$ that is true on all structures in *Pos* and false on all structures in *Neg*. We give an algorithm to synthesize such a sentence if one exists. Notice that since structures are finite, the sentence can also be converted to a program operating over structures that realizes the classifier effectively.

Automata over Parse Trees for Realizability and Synthesis. Our primary technique for solving exact learning problems of this kind is based on automata over finite trees. Intuitively, given finite (disjoint) sets of positive and negative structures, we need to search through an infinite set of formulas that adhere to the grammar G in order to find a separating formula. We use tree automata working over *formula parse trees* and show that the set of all separating formulas for *Pos* and *Neg* that adhere to G forms a regular class of trees. Building the tree automaton and checking emptiness gives us a decision procedure for realizability. Algorithms for tree automaton emptiness are used to solve the synthesis problem. Furthermore, the algorithms can be adapted to find the *smallest* trees that are accepted by the tree automaton, hence giving us small formulas as separators.

The key idea is to show that, given a single structure A, the set of parse trees for *all* sentences adhering to the grammar G that are true (or false) in A is a regular language. Given A, we can define an automaton that interprets an input formula (rather, its parse tree) on A and checks whether A satisfies the formula. This evaluation follows the usual semantics of the logic in question, which is typically defined *recursively*, and hence can be evaluated bottom-up in the structure of the formula. In general, this requires simulating the semantics of formulas for *each* assignment to free variables over the structure A. The bounded variable restriction is therefore crucial to ensure that the automaton only needs an amount of state that depends on the size of the structure but is *independent* of the size of the formula.

We can then construct an automaton that captures precisely the set of separating formulas for the given labeled structures. We do this by constructing automata for (a) the set of all formulas that are true on positive structures, (b) the set of all formulas that are false on negative structures, and (c) the intersection of the automata from (a) and (b). We further intersect (c) with an automaton that accepts formulas allowed by the grammar G. Checking emptiness of this final automaton solves realizability, and we can construct an accepted formula if nonempty.

Query and Term Synthesis. We study two related learning problems in addition to the separability problem. We study *query* synthesis, where we are given a grammar G and a finite set of structures, each accompanied by an *answer set* of r-tuples from the domain of the structure. The query synthesis problem is to find a query, namely, a first-order formula $\varphi \in L(G)$ with r free variables, such that the sets of tuples that satisfy φ in each structure are precisely the given answer sets. We also study the problem of *term synthesis*, which is closer in spirit to program synthesis from input-output examples (using logic as a programming language). In this problem we are given a set of input structures and a grammar G, and each structure interprets a set of constants, e.g., $in_1, \ldots, in_d$ and out, as a particular input-output example. The term synthesis problem is to find a closed first-order *term* t such that t evaluates to out in each structure. The grammar can ensure that out is not used in t. Again, we note that such first-order queries and terms can also be realized effectively as programs that operate over structures: a program for a query instantiates the free variables with all r-tuples, evaluates the formula, and returns those that satisfy it. Similarly, a term can be converted to a program that recursively evaluates its subterms and returns an element of the structure. We give adaptations of the automata-theoretic technique to solve both the query and term synthesis problems for the logic $\mathsf{FO}(k)$.

Learning Algorithms for First-Order Logic with Least Fixed Points. A second contribution of this work is showing that a bounded variable version of first-order logic with least fixed points also has decidable separator realizability and synthesis. Least fixed point definitions add the power of recursion to first-order logic, resulting in a more expressive logic that, for instance, can describe transitive closure of relations and the semantics of languages like Datalog. Furthermore, over finite ordered structures, first-order logic with least fixed points captures the class $\mathbb{P}$ of all functions computable in polynomial time. Consequently, learned formulas in this logic can be realized as polynomial-time programs.

In this case we use *two-way alternating tree automata* to succinctly encode the semantics of expressions with least fixed point definitions over a given structure. Intuitively, we need to use recursion to evaluate a recursively-defined relation, and in each recursive call we need to *read* the definition of the relation once more. This capability is elegantly provided by two-way automata, and alternation gives a way to compositionally send copies of the automaton to check various subformulas effectively. Two-way alternating automata can be converted to one-way nondeterministic automata (with an exponential increase in states), and emptiness checking for the resulting automata gives the algorithm we seek for separator realizability and synthesis. We also solve the term synthesis problem for a logic with least fixed points, a problem which resembles functional program synthesis.

Further Results. A remarkable aspect of the automata-theoretic approach is that it provides algorithms for synthesis in many settings. The constructions smoothly extend to virtually any logic where the bounded variable restriction yields a formula evaluation strategy with a memory requirement that is independent of the size of the formula. In §9 we discuss decidable realizability and synthesis results for other settings where related problems have been studied, including languages with mutual recursion, e.g., Datalog [Albarghouthi et al. 2017; Evans and Grefenstette 2018] and inductive logic programming [Cropper et al. 2020; Muggleton and de Raedt 1994; Muggleton et al. 2014]. We also discuss how the technique extends to higher-order logics over finite models.

Complexity. For each logic we consider, we present the upper bound that the automaton construction yields in terms of various parameters: the number of variables k, the maximum size of each structure, the number of structures, and the size of the grammar G. A sample of upper bounds is given in Table 1 in §4. We also prove lower bounds for the logic $\mathsf{FO}(k)$, arguing that the complexity of the upper bounds on certain parameters is indeed tight. In particular, we show that for fixed k, separator realizability is $\mathsf{EXPTIME}$-hard. This matches the upper bound, and also proves matching lower bounds for more expressive logics.

In summary, our work provides an extremely general tree automata-theoretic technique that yields effective solutions for the problems of learning separators/queries/terms for several finite-variable logics. Our contributions here are theoretical. We establish decidability for several exact learning problems over different logics, and we give algorithms based on tree automata and some matching lower bounds. We believe and hope that this work will inform the design of practical algorithms for these problems. In particular, our automata constructions and the “bottom up” fixed point procedure for checking automaton emptiness gives a design framework for such algorithms. Due to the relatively high worst-case complexity of synthesis, practical algorithms will need to adapt to application domains and cater to restricted logics and languages that admit more efficient synthesis, potentially using heuristics, space-efficient data structures, and fast search (e.g., BDDs, SAT solvers, etc.) (see [Bloem et al. 2012; Wang et al. 2017b, 2018] for examples).

Details for proofs and automata constructions can be found in the extended version [Krogmeier and Madhusudan 2021].

2 EXAMPLES

We begin with some examples to illustrate instances of the exact learning problems considered here. The first two examples explore the separability problem for first-order logic and the subtlety around reusing variables in bounded variable logics. The third example illustrates learning formulas with least fixed point definitions, and the fourth example illustrates term synthesis.

2.1 Example 1: Learning Formulas, Significance of Grammar, and Unrealizability

Consider the problem of finding a separating first-order sentence for the structures depicted in Figure 1 using a vocabulary that includes the binary edge relation E and constants s and t.

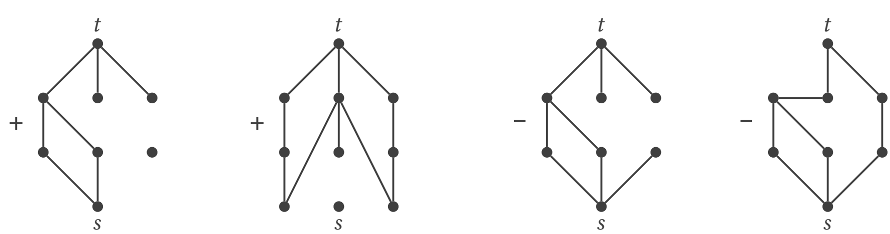


Fig. 1. Find a sentence in first-order logic that is true for + structures and false for − structures.

One possible solution asserts that *every node that is adjacent to* s *is adjacent to a node that is adjacent to* t. In first-order logic:

$$\forall x.\ (E(s,x) \rightarrow \exists y.\ (E(x,y) \wedge E(y,t)))$$

If the grammar G allowed, say, all first-order logic formulas with two variables, the formula above would indeed be a solution. If instead the grammar allowed only conjunction as a Boolean connective, then the formula above is not a separator. If the grammar allowed only one variable, or allowed two variables but disallowed universal quantification, then there is no separator.

In fact, for a grammar that only allows conjunction as Boolean connective, there is *no* separator for the structures above. *Proof gist.* For a contradiction: suppose there is a separator φ that does not use negation or disjunction. Then φ has a positive matrix (inner formula has no negations), and its standard conversion to an equivalent formula in prenex form, *prenex*(φ), still has a positive matrix and will be a separator (though it may have many more variables). Since *prenex*(φ) has a positive matrix, any graph that satisfies it will continue to satisfy it if we add any number of edges. Since the leftmost negative structure adds a single edge to the leftmost positive structure, we have a contradiction. Thus there is no separator.

Note that while algorithms can search for separators in G, the problem of declaring that there is *no* separator is a nontrivial problem, especially for arbitrary grammars. The algorithms we seek will terminate and declare unrealizability when separators do not exist (as in the above example).

2.2 Example 2: Reuse of Variables and Infinite Semantic Concept Space

Consider the problem of finding a separator that uses only *three variables* for the labeled structures in Figure 2. One possible solution is $\bigvee_{i=1}^{7} path_i(s,t)$, where $path_i(x,y)$ holds for elements x, y if there is a (directed) E-path of length i from x to y. Note that, by *reusing variables*, the formula $path_i(x,y)$ can be defined using only 3 variables for any $i \in \mathbb{N}$:

$$\begin{aligned} path_1(x,y) &\leftrightarrow E(x,y) \\ path_{i+1}(x,y) &\leftrightarrow \exists z.\, (E(x,z) \wedge \exists x.\, (x = z \wedge path_i(x,y))) \quad i > 0 \end{aligned}$$

This example shows that even when we restrict the number of variables, there is an infinite number of logically inequivalent sentences in first-order logic (e.g., $path_i$ for any $i > 0$). But note that this is not true if we bound the number of quantifiers or bound the depth of quantifiers. This infinite semantic space of concepts is what makes declaring unrealizability a nontrivial problem. As we will see, our technique works for finite variable logics in general, despite the fact that they admit infinitely-many inequivalent formulas.

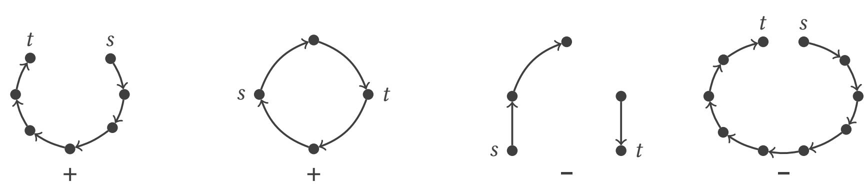


Fig. 2. Find a sentence in first-order logic that is true on + structures and false on − structures.

2.3 Example 3: Least Fixed Points and Recursive Definitions

Notice that the separating sentence from Figure 2 has a size that depends on the sizes of the input structures, and it fails to capture the notion of a path of *unbounded length.* Consider the problem in Figure 3. A separating concept is *all nodes can reach some cycle.* By augmenting first-order logic with (least fixed point) recursive definitions, this concept can be expressed using the following recursive definition for *reach* (which captures reachability using at least one edge):

$$\begin{aligned} \varphi \quad := \quad & \text{let } reach(x,y) =_{\mathsf{lfp}} E(x,y) \vee \exists z.\ (E(x,z) \wedge reach(z,y)) \\ & \text{in } \forall x.\ \exists y.\ (reach(x,y) \wedge reach(y,y)) \end{aligned}$$

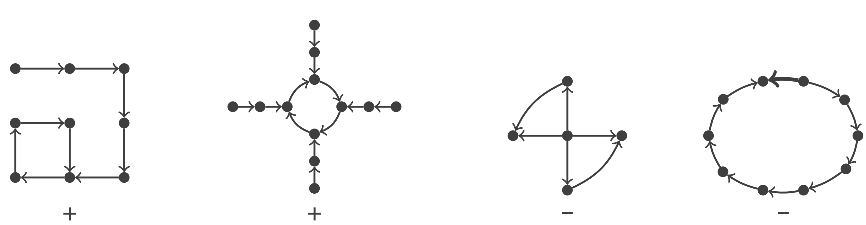


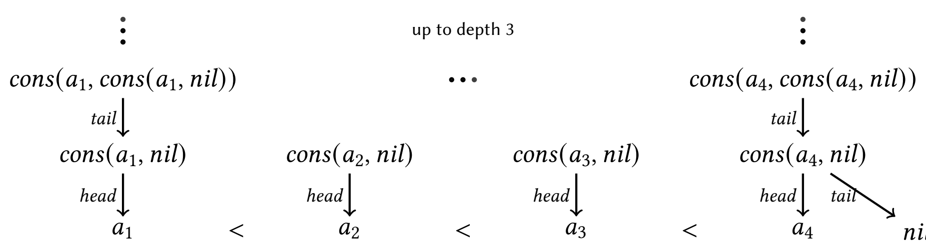
Fig. 3. Find a sentence in first-order logic with least fixed point definitions that is true on + structures and false on − structures and that does not depend on the sizes of the structures.

Recursive definitions dramatically increase the expressivity of first-order logic. As studied in finite model theory, such logics encompass (over structures equipped with a linear order on the domain) *all* polynomial-time computable properties, i.e., the class $\mathbb{P}$ [Immerman 1982; Libkin 2004; Vardi 1982]. We consider learning in a finite-variable version of first-order logic with least fixed points that captures all properties computable in time n^k using $O(k)$ variables. We note here a connection to the problem of synthesizing programs that are syntactically restricted in order to guarantee a specific *implicit complexity* [Dal Lago 2012], say polynomial time; we leave an exploration of this connection to future work.

2.4 Example 4: Term Synthesis and Program Synthesis

In addition to the separability problem illustrated in the previous examples, we also study the problem of *term synthesis*. In the term synthesis problem we aim to synthesize a term that evaluates to a specific element of the domain for each structure in a set of input structures. Specifically, given a set of structures, each with an interpretation for constants $in_1, \ldots, in_d$, and *out*, we want to construct a term t in the language of a grammar G such that t has the same interpretation as *out* in each structure. (Note the structures are not labeled in this problem.) The term synthesis problem, especially in the presence of recursive function definitions, resembles *functional program synthesis*.

Consider the problem of merging two sorted lists. We can model this setting with structures that represent finite prefixes of an abstract datatype for lists over a linearly ordered finite domain $\{a_1, a_2, \ldots, a_n\}$ (with ordering $<$). Figure 4 (top) depicts a portion of one such structure and its operations. We can model input-output tuples for the desired merge operation using an interpretation of constants in_1, in_2, and *out*, as depicted in the bottom of the figure.



Example	*in₁*	*in₂*	*out*
A_1	$cons(a_4, nil)$	$cons(a_2, cons(a_3, nil))$	$cons(a_2, cons(a_3, cons(a_4, nil)))$
A_2	$cons(a_1, cons(a_4, nil))$	$cons(a_3, nil)$	$cons(a_1, cons(a_3, cons(a_4, nil)))$

Fig. 4. (Top) Partial picture of a structure A_1 that encodes a finite prefix of a datatype for lists over an ordered domain, with terms bounded to depth 3. (Bottom) Input-output examples for *merge*. The goal is to find a closed term in first-order logic with least fixed point relations and recursive functions that evaluates to *out* on structures A_1 (top) and A_2 (not shown).

One possible solution is the following term t that defines a recursive function *merge* and applies it to the inputs in_1, in_2:

$$\begin{aligned} t \;\; := \;\; & \text{let } merge(x, y) =_{\text{lfp}} \; \text{ite}(x = nil, y, \text{ite}(y = nil, x, (\text{ite}(head(y) > head(x), \\ & \qquad\qquad\qquad\qquad cons(head(x), merge(tail(x), y)), \\ & \qquad\qquad\qquad\qquad cons(head(y), merge(x, tail(y))))))) \\ & \text{in } \; merge(in_1, in_2) \end{aligned}$$

3 BACKGROUND

We begin with some preliminary notions from logic as well as the concepts of *term*, *tree*, and *regular tree grammar*, which we will need for our definitions of various automata.

$$\varphi ::= R(\bar{t}) \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \neg\varphi \mid \exists x.\varphi \mid \forall x.\varphi \qquad t ::= x \mid c \mid f(\bar{t}) \mid \mathsf{ite}(\varphi, t, t')$$

Fig. 5. Grammar for first-order logic with if-then-else terms, denoted FO.

3.1 Logic

3.1.1 Structures and Signatures. A *first-order signature*, or simply *signature*, is a set τ of sets of relation symbols $\{R_1, R_2, \ldots\}$, function symbols $\{f_1, f_2, \ldots\}$, and constant symbols $\{c_1, c_2, \ldots\}$. Each symbol s has an associated arity, denoted $\mathsf{arity}(s) \in \mathbb{N}$. The meaning of symbols in a signature depends on a *τ-structure*, which is a tuple $A = \langle \mathsf{dom}(A), R_1^A, \ldots, R_a^A, f_1^A, \ldots, f_b^A, c_1^A, \ldots, c_c^A \rangle$. The domain $\mathsf{dom}(A)$ is a set, each R_i^A is a relation on the domain, i.e., $R_i^A \subseteq \mathsf{dom}(A)^{\mathsf{arity}(R_i)}$, each f_j^A is a total function on the domain, i.e., $f_j^A : \mathsf{dom}(A)^{\mathsf{arity}(f_j)} \longrightarrow \mathsf{dom}(A)$, and each constant c denotes an element $c^A \in \mathsf{dom}(A)$. For simplicity, we model constants as nullary functions. Each problem addressed in this work involves *finite* structures, i.e., those for which $|\mathsf{dom}(A)| \in \mathbb{N}$. Thus *structure* will always mean *finite structure*. We omit τ and write *structure* whenever τ can be understood from context or is unimportant. We use A to denote an arbitrary structure.

3.1.2 First-Order Logic. Though the technique presented in this work is highly versatile, we will focus the majority of our presentation on variants and extensions of first-order logic. As a starting point, we consider first-order logic extended with an if-then-else term. Syntax for this logic, denoted FO, is given in Figure 5. The semantics of the usual FO formulas and terms is standard. We denote the interpretation of a term t in a structure A and variable assignment γ as $t^{A,\gamma}$. The interpretation of the if-then-else term in A, γ, is:

$$\mathsf{ite}(\varphi, t_1, t_2)^{A,\gamma} = \begin{cases} t_1^{A,\gamma} & A, \gamma \models \varphi \\ t_2^{A,\gamma} & \text{otherwise} \end{cases}$$

We use $\mathcal{L}$ to refer to an arbitrary logic, and occasionally, if we want to emphasize the signature we write $\mathcal{L}(\tau)$ for a logic $\mathcal{L}$ over τ. See [Enderton 2001] for syntax, semantics, and basic results in first-order logic.

3.2 Terms and Trees

Rather than working with strings, it will be simpler to instead consider logical formulas as finite ordered ranked trees, sometimes called *terms*. Intuitively, to build terms we use symbols from a finite *ranked alphabet*, that is, a set of symbols with corresponding arities. We use $T_\Sigma(X)$ to denote the set of terms over a ranked alphabet Σ augmented with nullary symbols X (with X disjoint from Σ). When $X = \emptyset$ we just write T_Σ.

It will be convenient to also use the language of ordered trees. An *ordered tree* ρ over a label set W is a partial function $\rho : \mathbb{N}^* \longrightarrow W$ defined on $\mathit{Nodes}(\rho) \subseteq \mathbb{N}^*$, a prefix-closed set of positions containing a root $\epsilon \in \mathit{Nodes}(\rho)$. In this view, terms are simply ordered trees whose labels respect ranks, that is, ordered trees subject to the following requirement: if $a \in \Sigma$ with $\mathsf{arity}(a) = n$ and for some $x \in \mathit{Nodes}(\rho)$ we have $\rho(x) = a$, then $\{j \in \mathbb{N} \mid x \cdot j \in \mathit{Nodes}(\rho)\} = \{1, \ldots, n\}$. We will refer to ordered (ranked) trees as simply *trees* to avoid confusion with the usual syntactic category of *logical terms*.

3.3 Finite Variable Logics to Trees

When can the formulas of a logic be represented as trees over a finite alphabet? We probably must have a finite signature, as well as syntax formation rules that take a finite number of subformulas. For any logic $\mathcal{L}$ and signature τ that meets these requirements, if we bound the number of variables

$$S \;\rightarrow\; \vee(S,S) \mid \wedge(S,S) \mid \neg(S) \mid \exists x(S) \mid \forall x(S) \mid \exists y(S) \\ \mid \forall y(S) \mid E(x,x) \mid E(x,y) \mid E(y,x) \mid E(y,y)$$

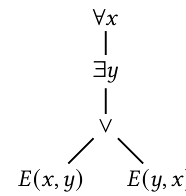


Fig. 6. (Left) Production rules from the set P for a regular tree grammar $G = \langle \{S\}, \Sigma_{\mathcal{L}}, S, P \rangle$, where $\mathcal{L}$ is $\mathsf{FO}(k)(\mathsf{graph})$ with $k = 2$, and (Right) a tree in $L(G)$ for the sentence $\forall x.\exists y.\, (E(x,y) \vee E(y,x))$.

that can appear in any formula, then we can define a *finite* ranked alphabet $\Sigma_{\mathcal{L}(\tau)}$ such that any formula $\varphi \in \mathcal{L}(\tau)$ has at least one corresponding tree $t \in T_{\Sigma_{\mathcal{L}(\tau)}}$. For example, consider a variant of FO restricted to the k variables in $V = \{x_1, \ldots, x_k\}$, which we denote $\mathsf{FO}(k)$[1]. With superscripts for arity, the ranked alphabet looks as follows:

$$\Sigma_{\mathsf{FO}(k)} = \left\{R^{\mathsf{arity}(R)} \;\middle|\; R \in \tau\right\} \cup \left\{f^{\mathsf{arity}(f)} \;\middle|\; f \in \tau\right\} \cup \left\{\mathsf{ite}^3\right\} \cup \left\{\wedge^2, \vee^2, \neg^1\right\} \cup \left\{\forall x^1, \exists x^1, x^0 \mid x \in V\right\}$$

We sometimes drop the subscript for the underlying logic and just use Σ to refer to finite ranked alphabets of this kind.

3.4 Regular Tree Grammars

A regular tree grammar (RTG) is a tuple $G = \langle N, \Sigma, S, P \rangle$, where N is a finite set of nonterminal symbols, Σ is a finite ranked alphabet, $S \in N$ is the axiom nonterminal, and P is a set of rewrite rules of the form $B \longrightarrow t$, where $B \in N$ and $t \in T_\Sigma(N)$. The language $L(G)$ of G is the set of trees $\{t \in T_\Sigma \mid S \Rightarrow^* t\}$, where $t \Rightarrow t'$ holds whenever there is a context C and tree $t'' \in T_\Sigma(N)$ such that $t = C[B], t' = C[t'']$ and $B \longrightarrow t'' \in P$. These are standard notions; see [Comon et al. 2007; Grädel et al. 2002] for details.

Given a logic $\mathcal{L}$, we consider RTGs *over* $\mathcal{L}$. If $\Sigma_{\mathcal{L}}$ is a finite ranked alphabet for $\mathcal{L}$, then an RTG over $\mathcal{L}$ is of the form $G = \langle N, \Sigma_{\mathcal{L}}, S, P \rangle$ for some N, S, P. In Figure 6 we give an example RTG over $\mathsf{FO}(k)(\mathsf{graph})$ with $k = 2$, i.e., first-order logic with variables $V = \{x, y\}$ over a signature graph consisting of a single binary relation symbol E. When we refer to sentences or formulas in the remainder of this paper we mean their corresponding trees in a suitable RTG, and when we refer to a *grammar* we mean an RTG. When there are multiple ways to represent a given formula as a tree, we pick one arbitrarily.

3.5 Alternating Tree Automata

As we will show, the formalism of *alternating tree automata* yields an elegant technique for evaluating expressions on a fixed structure. We summarize the relevant ideas from automata theory.

An alternating tree automaton (ATA) over κ-ary trees is a tuple $\mathcal{A} = \langle Q, \Sigma, I, \delta \rangle$, where Q is a finite set of states, Σ is a finite ranked alphabet, $I \subseteq Q$ is a set of initial states, and the transition function has the form $\delta : Q \times \Sigma \longrightarrow \mathcal{B}^+(Q \times \{1, \ldots, \kappa\})$, where $\mathcal{B}^+(X)$ denotes the set of positive propositional formulas over atoms from a set X. For any $(q, a) \in Q \times \Sigma$ we require $\delta(q, a) \in \mathcal{B}^+(Q \times \{1, \ldots, \mathsf{arity}(a)\})$. For example, if $f \in \Sigma$ and $\mathsf{arity}(f) = 2$, we might have $\delta(q, f) = (q_1, 1) \wedge (q_2, 2) \vee (q_1', 1) \wedge (q_2', 2)$. This transition stipulates that, when reading the symbol f in state q, the automaton must *either* successfully continue from the left child in state q_1 and from the right child in state q_2 *or* it must successfully continue from the left child in state q_1' and from

[1]We overload notation, using $\mathsf{FO}(\tau)$ for FO over signature τ and $\mathsf{FO}(k)$ for FO with k variables and an unspecified signature. If the two notations are both needed at once we put the signature last, e.g., $\mathsf{FO}(k)(\tau)$ is FO with k variables over τ.

the right child in state q_2'. This, in fact, is already expressible as a *nondeterministic tree automaton* transition. A nondeterministic tree automaton can be viewed as an ATA whose transition formulas are in disjunctive normal form, where each conjunctive subformula refers to each child at most once. What alternation buys is a transition like, e.g., $\delta(q, f) = (q_1, 1) \wedge (q_2, 2) \wedge (q_1', 1) \wedge (q_2', 2)$, in which there are multiple distinct conditions placed on a single child.

The *language* $L(\mathcal{A})$ of an ATA $\mathcal{A}$ is the set of trees that it accepts. This set is defined with respect to a *run*, which captures the idea of a pass over an input tree that succeeds in satisfying the conditions stipulated by the transition function. For the automata in this work, some transitions will use the formula False, and trees in the language of such an automaton can be thought of as those which are able to satisfy the transition formulas in such a way that they never are forced to satisfy False, which is impossible.

A *run* of an ATA $\mathcal{A} = \langle Q, \Sigma, I, \delta \rangle$ on an input $t \in T_\Sigma$ is an ordered tree ρ over the label set $Q \times \mathit{Nodes}(t)$ satisfying the following two conditions:

- $\rho(\epsilon) = (q_i, \epsilon)$ for some state $q_i \in I$
- Let $n \in \mathit{Nodes}(\rho)$. If $\rho(n) = (q, x)$ with $t(x) = a$, then there exists $S = \{(q_1, i_1), \ldots, (q_l, i_l)\} \subseteq Q \times \{1, \ldots, \text{arity}(a)\}$ such that $S \models \delta(q, a)$ and $\rho(n \cdot j) = (q_j, x \cdot i_j)$ for $1 \leq j \leq l$.

An ATA $\mathcal{A}$ *accepts* a tree t if it has a run on t. Note the structure of a run ρ can be different from that of the input t, since it records how the automaton satisfies the transition function, which can involve going to several states for any given child. This is important for the complexity of emptiness checking for ATAs, because it means that mere reachability of states is not enough to verify a transition can be taken: for some transitions one must also verify that reachability of certain states is *witnessed by the same tree*. We note that (1) an ATA can be converted to a nondeterministic tree automaton with the same language in exponential time (incurring an exponential increase in states) and (2) emptiness for nondeterministic tree automata is decidable in linear time.

We will use $L(\mathcal{A}, q)$ to denote the language of an automaton $\mathcal{A}$ when we view $q \in Q$ as an initial state (thus $L(\mathcal{A}) = \cup_{q \in I} L(\mathcal{A}, q)$). We refer the reader to [Comon et al. 2007] for details and standard results about tree automata.

4 REALIZABILITY AND SYNTHESIS PROBLEMS

In this section we define three exact learning problems that are parameterized by a logic $\mathcal{L}$. The first problem involves separating a set of labeled structures using a sentence in $\mathcal{L}$. The second problem involves finding a formula in $\mathcal{L}$ that exactly defines a given set of tuples over the domain of a structure. The third problem involves finding a term in $\mathcal{L}$ that obtains specified domain values in given structures. For each of these problems, we always assume a fixed and finite signature τ.

The first problem, *$\mathcal{L}$-separator realizability and synthesis*, is defined in Problem 1. Given positively- and negatively-labeled structures and a grammar G over $\mathcal{L}$, the problem is to synthesize a sentence in G such that all positive structures make the sentence true and all negative structures make it false, or declare no such sentence exists. Sometimes we refer to this as the *separability* problem.

Problem 1: *$\mathcal{L}$-separator realizability and synthesis*

Input: $\left\langle \mathit{Pos} = \{A_1, \ldots, A_{m_1}\}, \mathit{Neg} = \{B_1, \ldots, B_{m_2}\}, G \right\rangle$ where
A_i, B_j are τ-structures
G an RTG over $\Sigma_{\mathcal{L}}$

Output: $\varphi \in L(G)$ *s.t.* for all $A_i \in \mathit{Pos}$, $A_i \models \varphi$, and for all $B_j \in \mathit{Neg}$, $B_j \not\models \varphi$
Or "No" if no such φ exists

Table 1. Summary of main results for a fixed signature and fixed arities of relations and functions.

<table>
<tr><th>Problem</th><th>Parameters</th><th>Time complexity</th><th>Combined complexity (fixed variables)</th></tr>
<tr><td>FO separability</td><td rowspan="3">m input structures
n max structure size
k first-order variables</td><td rowspan="3">$O\left(2^{poly\left(mn^k\right)}\lvert G\rvert\right)$</td><td rowspan="5">EXPTIME-complete in $mn + \lvert G\rvert$</td></tr>
<tr><td>FO queries</td></tr>
<tr><td>FO term synthesis</td></tr>
<tr><td>FO-LFP separability</td><td>FO parameters *and* k' relation variables</td><td>$O\left(2^{poly\left(mn^k k'\right)}\lvert G\rvert\right)$</td></tr>
<tr><td>FO-LFP-FUN term synthesis</td><td>FO parameters *and*
k_1 relation variables
k_2 function variables</td><td>$O\left(2^{poly\left(mn^k (k_1+k_2)\right)}\lvert G\rvert\right)$</td></tr>
</table>

The second problem involves synthesizing r-ary *queries*. A r-ary query for a logic $\mathcal{L}$ is a formula $\varphi(x_1, \ldots, x_r) \in \mathcal{L}$ that has exactly r distinct free variables, all first-order. The *answer set* for a r-ary query φ in a structure A is the precise set of tuples $Ans = \{\, \bar{a} \in \mathrm{dom}(A)^r \mid A \models \varphi(\bar{a}) \,\}$ that make the query true in the structure. For example, consider a "family relationships" domain with two structures A_1 and A_2. In A_1 there are domain elements *Sue* and *Bob* and the relationship Mother(*Sue*, *Bob*), and in A_2 there are elements *Maria*, *Tom*, and *Anne* and the relationships Mother(*Maria*, *Tom*) and Mother(*Maria*, *Anne*). Suppose the answer sets are $Ans_1 = \{Sue\}$ and $Ans_2 = \{Maria\}$. Then one possible solution is the query $\varphi(x) := \exists y.\mathrm{Mother}(x, y)$.

We call this second problem *$\mathcal{L}$-query realizability and synthesis*, which is defined formally in Problem 2. Given a grammar G over $\mathcal{L}$ and a set of pairs, where each pair is a structure and an answer set, synthesize a query φ in G such that φ precisely defines the given answer set in each structure, or declare no such φ exists.

Problem 2: *$\mathcal{L}$-query realizability and synthesis*

Input: $\langle \{\langle A_1, Ans_1\rangle, \ldots, \langle A_m, Ans_m\rangle\}, G\rangle$ where
A_i are τ-structures
$Ans_i \subseteq \mathrm{dom}(A_i)^r$
G an RTG over $\Sigma_{\mathcal{L}}$

Output: $\varphi(x_1, \ldots, x_r) \in L(G)$ *s.t.* $\{\bar{a} \in \mathrm{dom}(A_i)^r \mid A_i \models \varphi(\bar{a})\} = Ans_i$ for all $i \in [m]$
Or "No" if no such φ exists

The third problem, *$\mathcal{L}$-term synthesis*, is defined in Problem 3. The input is a grammar G and a set of (unlabeled) structures $\{A_1, \ldots, A_m\}$. Each structure A_i interprets constants $in_1, \ldots, in_d$, and *out*, where *out* is the *target element* in the domain of each structure. The goal is to synthesize a term t from G (which precludes using *out*) such that $A_i \models (t = out)$ for each i.

In Table 1 we highlight our main results for these three problems instantiated with various logics. Note that the upper bounds on time complexity assume a fixed signature, and in particular, fixed arities of symbols. The remainder of the paper lays out our general automata-theoretic solution.

5 SOLVING REALIZABILITY AND SYNTHESIS FOR FIRST-ORDER LOGIC

In this section we describe our general technique by instantiating it on the separability problem for the logic FO(k) (§5.1). We then show how to adapt the solution to solve query synthesis for the same logic (§5.2). Term synthesis is covered in §7.

Problem 3: $\mathcal{L}$-*term synthesis*

Input: $\langle\{A_1, \ldots, A_m\}, G\rangle$ where
A_i are τ-structures
G an RTG over $\Sigma_{\mathcal{L}}$
Output: $t \in L(G)$ $s.t.$ $A_i \models (t = out)$ for all $i \in [m]$
Or "No" if no such t exists

5.1 Separator Realizability and Synthesis in First-Order Logic

Consider separability for $\mathsf{FO}(k)$ over an arbitrary signature. We are given a grammar G and sets of positive and negative structures *Pos* and *Neg*. The main idea is to build an alternating tree automaton that accepts the parse trees of *all* formulas that separate *Pos* and *Neg*. This automaton itself is constructed as the product of automata $\mathcal{A}_M$, one for each structure $M \in Pos \uplus Neg$. If $M \in Pos$, then $\mathcal{A}_M$ accepts all formulas that are true on M. If $M \in Neg$, then $\mathcal{A}_M$ accepts all formulas that are false on M. Clearly, the intersection of these automata gives the desired automaton $\mathcal{A}_\cap$ that accepts formulas which separate *Pos* and *Neg*. In §5.1.1, we give the main construction of $\mathcal{A}_M$ for each $M \in Pos \uplus Neg$, which involves evaluating a given input formula on a fixed structure M.

We build another tree automaton $\mathcal{A}_G$ (§5.1.2) that accepts precisely the formulas from G, and finally we construct an automaton accepting the intersection of languages for $\mathcal{A}_\cap$ and $\mathcal{A}_G$ (§5.1.3). Checking emptiness of this automaton solves the realizability problem and, when the language is nonempty, finding a member of the language solves the synthesis problem.

5.1.1 Automaton for Evaluating First-Order Logic Formulas. We now show how to construct a tree automaton that accepts the set of sentences in $\mathsf{FO}(k)$ that are true in a given structure A. For clarity, we present an automaton for a slightly simpler version of $\mathsf{FO}(k)$ over an arbitrary relational signature τ and without an if-then-else term (thus the only terms are variables). The ranked alphabet for this simplification over $\tau = \langle R_1, \ldots, R_s\rangle$ is:

$$\Sigma'_{\mathsf{FO}(k)} = \left\{R_i(\bar{x})^0 \;\middle|\; R_i \in \tau, \bar{x} \in V^{\mathsf{arity}(R_i)}\right\} \cup \left\{\wedge^2, \vee^2, \neg^1\right\} \cup \left\{\forall x^1, \exists x^1 \mid x \in V\right\}$$

Note that each atomic formula over variables V becomes a nullary symbol. Handling the full gamut of terms in FO from Figure 5 is straightforward but tedious, so we omit the details. Following the simpler construction, we give the high-level idea for the full version.

Fix a τ-structure A with $|\mathsf{dom}(A)| = n$. We define an ATA $\mathcal{A}_A = \langle Q, \Sigma'_{\mathsf{FO}(k)}, I, \delta\rangle$ whose language is the set of trees over $\Sigma'_{\mathsf{FO}(k)}$ corresponding to sentences that are true in the structure A. Each component is discussed below.

States. The states of $\mathcal{A}_A$ are partial assignments from variables $V = \{x_1, \ldots, x_k\}$ to the domain $\mathsf{dom}(A)$. We denote the set of partial assignments by $Assign \coloneqq V \rightharpoonup \mathsf{dom}(A)$, and we use γ to range over *Assign*. States of the automaton keep track of assignments that accrue when the automaton reads quantification symbols. The crucial idea is that for each syntax formation rule, we can express the conditions under which the formula is true with the current assignment as a positive Boolean formula over assignments and subformulas. The only hiccup is that the automaton needs to keep track of whether or not a formula should be satisfied or *not* satisfied, which is dictated by occurrences of negation. We need a single bit for this, and so the state space increases by a factor of two. A state $\gamma \in Assign$ can be marked $\tilde{\gamma}$ to indicate that under assignment γ the input formula should *not* be true. We use $Dual(X) \triangleq \{x, \tilde{x} \mid x \in X\}$ to denote a set X together with marked copies of its elements. With this notation, the state set for our automaton is $Q \coloneqq Dual(Assign)$, and $|Q| = O(n^k)$. For a given γ we abuse notation and treat γ as a set of variable-binding pairs, for instance, $\{x \mapsto a_1, y \mapsto a_2\}$ and $\emptyset$ denote assignments in this way. We use $\gamma[x \mapsto a]$ to denote the assignment that is identical to γ except it maps x to a. We write $\gamma(\bar{x}) \downarrow$ to denote that γ is defined on each $x_i \in \bar{x}$ and $\gamma(\bar{x})$ to denote the tuple of elements obtained by applying γ to $\bar{x}$.

Initial states. There is only one initial state, namely, the one that assigns no variables: $I = \{\emptyset\}$.

Transitions. To define the transition function, for each assignment $\gamma \in \mathit{Assign}$ and each symbol $a \in \Sigma'_{\mathsf{FO}(k)}$, we give a propositional formula that naturally mimics the semantics of first-order logic. The intuition is that, from a state $\gamma \in \mathit{Assign}$, the automaton accepts every formula that is true in the structure A when free variables are interpreted according to γ. For $\gamma \in \mathit{Assign}$ and $x, \bar{x}$ ranging over V, the transitions are as follows:

$$\delta(\gamma, \wedge) = (\gamma, 1) \wedge (\gamma, 2) \qquad \delta(\tilde{\gamma}, \wedge) = (\tilde{\gamma}, 1) \vee (\tilde{\gamma}, 2)$$

$$\delta(\gamma, \vee) = (\gamma, 1) \vee (\gamma, 2) \qquad \delta(\tilde{\gamma}, \vee) = (\tilde{\gamma}, 1) \wedge (\tilde{\gamma}, 2)$$

$$\delta(\gamma, \forall x) = \bigwedge\nolimits_{a \in \mathsf{dom}(A)} (\gamma[x \mapsto a], 1) \qquad \delta(\tilde{\gamma}, \forall x) = \bigvee\nolimits_{a \in \mathsf{dom}(A)} (\tilde{\gamma'}, 1), \quad \gamma' = \gamma[x \mapsto a]$$

$$\delta(\gamma, \exists x) = \bigvee\nolimits_{a \in \mathsf{dom}(A)} (\gamma[x \mapsto a], 1) \qquad \delta(\tilde{\gamma}, \exists x) = \bigwedge\nolimits_{a \in \mathsf{dom}(A)} (\tilde{\gamma'}, 1), \quad \gamma' = \gamma[x \mapsto a]$$

$$\delta(\gamma, R(\bar{x})) = \begin{cases} \mathsf{True} & \gamma(\bar{x}) \downarrow,\ A, \gamma \models R(\bar{x}) \\ \mathsf{False} & \text{otherwise} \end{cases} \qquad \delta(\tilde{\gamma}, R(\bar{x})) = \begin{cases} \mathsf{True} & \gamma(\bar{x}) \downarrow,\ A, \gamma \not\models R(\bar{x}) \\ \mathsf{False} & \text{otherwise} \end{cases}$$

$$\delta(\gamma, \neg) = (\tilde{\gamma}, 1) \qquad \delta(\tilde{\gamma}, \neg) = (\gamma, 1)$$

Note that for all $(q, a) \in Q \times \Sigma'_{\mathsf{FO}(k)}$ we have $\delta(q, a) \in \mathcal{B}^+(Q \times [\mathsf{arity}(a)])$, where for nullary symbols we can take $[0] = \emptyset$.

Lemma 1. $\mathcal{A}_A$ accepts any sentence φ over $\Sigma'_{\mathsf{FO}(k)}$ that is true in A.

Proof Sketch. A simple induction shows that for each assignment γ (resp. $\tilde{\gamma}$), the language of $\mathcal{A}_A$ from that state is precisely the set of formulas that are true (resp. false) in A under γ, i.e., $L(\mathcal{A}_A, \gamma) = \{\varphi(\bar{x}) \mid \gamma(\bar{x}) \downarrow,\ A, \gamma \models \varphi(\bar{x})\}$ (resp. $L(\mathcal{A}_A, \tilde{\gamma}) = \{\varphi(\bar{x}) \mid \gamma(\bar{x}) \downarrow,\ A, \gamma \not\models \varphi(\bar{x})\}$). □

If we fix an ordering on variables, then we can identify a state γ in the obvious way with a tuple of domain elements $\bar{a}$. Then the language of the automaton at γ coincides with the notion of *logical type* for the pair $(A, \bar{a})$ [Libkin 2004]. One consequence of this generality is that the automaton can be seamlessly adapted to solve the query problem for $\mathsf{FO}(k)$, as we will see in §5.2.

5.1.2 Grammar Automaton. As the name suggests, the language of an RTG is regular, and so it is the language of some tree automaton. Given $G = \langle N, \Sigma, S, P\rangle$ an RTG, a (nondeterministic) tree automaton for it is simple to define. We let $\mathcal{A}_G = \langle N, \Sigma, S, \delta\rangle$, where for each $B \to f(B_1, \ldots, B_{\mathsf{arity}(f)}) \in P$ we set $\delta(B, f) = \bigwedge_i (B_i, i)$. Observe that we have not used the full power of alternation: the transition function puts at most one condition on any given child. Thus the automaton is already nondeterministic, which keeps the size of the final automaton (§5.1.3) linear in the size of the grammar. Notice also that we have made a simplifying assumption about the form of rules in P, since the right-hand side could contain subterms that are not nonterminal symbols. It is easy to show that more complicated rules can be represented using multiple simple rules over a larger state space, yielding an equivalent grammar of size $O(|G|)$.

5.1.3 Decision Procedure. The decision procedure for realizability and synthesis is as follows. We define the automaton $\mathcal{A}_A$ as described above for each structure $A \in \mathit{Pos}$, as well as the automaton $\mathcal{A}_G$ for the grammar G. For each structure $B \in \mathit{Neg}$ we define $\mathcal{A}_B$ in the same way as for positive structures, with one tweak: instead of initial states $I = \{\emptyset\}$ we have $I = \{\tilde{\emptyset}\}$. We take the product of the structure automata to get:

$$\mathcal{A}_\cap = \bigtimes_{M \in Pos \uplus Neg} \mathcal{A}_M$$

with number of states $O(mn^k)$, where $m = |Pos \uplus Neg|$, $n = \max_{M \in Pos \uplus Neg} |\mathrm{dom}(M)|$, and k is the number of variables. There is an exponential increase in states to convert $\mathcal{A}_\cap$ to a nondeterministic automaton $\mathcal{A}'_\cap$ with $L(\mathcal{A}'_\cap) = L(\mathcal{A}_\cap)$ [Comon et al. 2007; Grädel et al. 2002]. Finally, we take the product of $\mathcal{A}'_\cap$ and $\mathcal{A}_G$ to get $\mathcal{A} = \mathcal{A}'_\cap \times \mathcal{A}_G$, with $L(\mathcal{A}) = L(\mathcal{A}'_\cap) \cap L(\mathcal{A}_G)$, and furthermore, $L(\mathcal{A}) \neq \emptyset$ if and only if there is a sentence $\varphi \in L(G)$ that separates the input structures. We solve realizability by checking emptiness of $\mathcal{A}$ in time linear in its size, which is $O(2^{poly(mn^k)} |G|)$, and the emptiness checking algorithm can construct a (small) tree if nonempty.

This construction can be easily extended to give us the following theorem for realizability and synthesis in the full logic $\mathsf{FO}(k)$ that includes if-then-else and function terms.

Theorem 2. $\mathsf{FO}(k)$-separator realizability and synthesis *is decidable in* EXPTIME *for a fixed signature and fixed* $k \in \mathbb{N}$.

The construction for the full gamut of terms is straightforward. It can be accomplished by not only keeping partial assignments but also states that encode the currently expected domain element for a term under evaluation by the automaton. We give more details for how this extension works when we discuss term synthesis in §7.

5.2 Query Realizability and Synthesis

As noted, the automaton $\mathcal{A}_A$ (§5.1.1) is more general than an acceptor of sentences, and it can be easily modified as follows to solve query synthesis for $\mathsf{FO}(k)$ with no increase in complexity.

For a pair $\langle A, Ans \rangle$ of a structure and an answer set, with $Ans \subseteq \mathrm{dom}(A)^r$, we define an ATA $\mathcal{A}_A = \langle Q, \Sigma, I, \delta \rangle$ whose language is the set of all formulas $\varphi(y_1, \ldots, y_r) \in \mathsf{FO}(k)$, with $r \leq k$, whose answer set in A is Ans. We describe each component below. For simplicity, we work with a fixed permutation of r distinct variables $\bar{y} \in V^r$, where $V = \{x_1, \ldots, x_k\}$.

States. The set of states is unchanged: $Q := Dual(Assign)$.

Transitions. The transition function δ is unchanged, with the exception of the following transitions for the initial state $q_i = \emptyset$.

Initial states. $I = \{q_i\}$. The main idea is to (a) require that the automaton reads and accepts the input formula from all states (partial variable assignments) that correspond to tuples in the answer set Ans, and (b) reject from states corresponding to the complement of Ans. Let $S(\bar{y}) \subseteq Q$ be the set of assignments defined only on $\bar{y}$, and let $S(Ans) \subseteq S(\bar{y})$ be the subset of assignments that map $\bar{y}$ to a member of the answer set. For any $a \in \Sigma'_{\mathsf{FO}(k)}$, the transition out of q_i is given by:

$$\delta(q_i, a) \quad = \quad \left(\bigwedge_{\gamma \in S(Ans)} \delta(\gamma, a) \right) \wedge \left(\bigwedge_{\gamma \in S(\bar{y}) \setminus S(Ans)} \delta(\tilde{\gamma}, a) \right)$$

The following theorem follows easily from the proof of Lemma 1.

Theorem 3. $\mathsf{FO}(k)$-query realizability and synthesis *is decidable in* EXPTIME *for a fixed signature and fixed* $k \in \mathbb{N}$.

6 REALIZABILITY AND SYNTHESIS WITH LEAST FIXED POINT DEFINITIONS

In this section we study separability for logics with least fixed point operators. In particular, we choose a logic with a finite set of *relation variables*, each of which can be defined recursively. These relation variables, though finite in number, can be redefined any number of times (similar to reusing variables, as we saw in §2.2). Further, as we will discuss in §9.1, the ideas presented in this section can be extended to handle mutually recursive definitions.

Note that the ability to *define* a relation is valuable independently of recursion: with definitions we can require a formula to be synthesized and then used in *multiple distinct places*. For example, we may want to express: *there exist x and y that are related in some unknown way, and further, all things related in that way also share a property* ψ. In logic, this amounts to a separator of the form:

$$\exists x. \exists y.\, \varphi(x,y) \land (\forall x. \forall y.\, \varphi(x,y) \rightarrow \psi(x,y))$$

Notice that φ appears twice, which we cannot express with a regular tree grammar. However, with relation variables and definitions we can ask to synthesize a formula φ in a template as follows:

$$\mathsf{let}\ R(x,y) = \varphi(x,y)\ \mathsf{in}\ \exists x. \exists y.\, R(x,y) \land (\forall x. \forall y.\, R(x,y) \rightarrow \psi(x,y))$$

Following the semantics of our logic with least fixed point definitions, we will see how the automata-theoretic approach extends neatly to accommodate both definitions and recursion by moving from alternating tree automata to two-way tree automata.

6.1 First-Order Logic with Least Fixed Points

Here we describe FO-LFP, which is an extension of FO with recursively-defined relations with least fixed point semantics. The syntax is given in Figure 7. Formulas in FO-LFP can define relations using a set $\{P_1, P_2, \ldots\}$ of symbols disjoint from the signature. Such symbols are interpreted as least fixed points of the set operators induced by their definitions. Note that, although not shown in Figure 7, we require all relations to be defined before they are used.

Recall the definition for *reachability* from Figure 3:

$$\varphi := \ \mathsf{let}\ \mathit{reach}(x,y) =_{\mathsf{lfp}} (E(x,y) \lor \exists z.\ E(x,z) \land \mathit{reach}(z,y))\ \ \mathsf{in}\ \ \varphi'(\mathit{reach})$$

For a fixed structure A, the meaning of $\mathit{reach}(x,y)$ is obtained by first interpreting the definition $\psi(x,y,\mathit{reach}) := E(x,y) \lor \exists z.\ E(x,z) \land \mathit{reach}(z,y)$ as a monotonic function $F_\psi : 2^X \rightarrow 2^X$ over the lattice defined by the subset relation on 2^X, where $X = \mathsf{dom}(A)^2$. Formally, for $Y \subseteq X$,

$$F_\psi(Y) \triangleq \{(a_1,a_2) \in X \mid \psi(a_1/x, a_2/y, Y/\mathit{reach})\},$$

where by (Y/reach) we mean that *reach* is interpreted as the relation Y in ψ (similarly for a_1/x and a_2/y). Then for any $a, a' \in \mathsf{dom}(A)$, $\mathit{reach}(a,a')$ holds if and only if $(a,a') \in \mathsf{lfp}(F_\psi)$, where $\mathsf{lfp}(F_\psi)$ denotes the least fixed point of F_ψ. More generally, definitions in FO-LFP are interpreted as follows in a structure A:

$$A \models \mathsf{let}\, P(\bar{x}) =_{\mathsf{lfp}} \psi(\bar{x},P)\ \mathsf{in}\ \varphi(P) \quad \Leftrightarrow \quad A \models \varphi(\mathsf{lfp}(F_\psi)/P) \tag{1}$$

Note that the least fixed point $\mathsf{lfp}(F_\psi)$ may not exist for an arbitrary formula ψ. It turns out, however, that a simple syntactic restriction can ensure existence of least fixed points. Technically, we require all occurrences of P in ψ to occur under an even number of negations. This restriction can be enforced by the grammar, and we will not mention it further.

We consider a variant of FO-LFP with a finite number of recursive relation symbols P, but the symbols can be reused in later definitions. That is, they can be shadowed. Our semantics for definitions therefore assumes that defined relations are renamed uniquely before applying Equation (1). The semantics for the rest of FO-LFP is straightforward and follows that of FO.

$$\begin{array}{lcl} \varphi & ::= & \mathsf{let}\ P(\bar{x}) =_{\mathsf{lfp}} \psi\ \mathsf{in}\ \varphi \\ & & \mid \exists x.\varphi \mid \forall x.\varphi \mid R(\bar{t}) \mid P(\bar{t}) \\ & & \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \neg\varphi \end{array} \qquad \begin{array}{lcl} \psi & ::= & \exists x.\psi \mid \forall x.\psi \mid R(\bar{t}) \mid P(\bar{t}) \\ & & \mid \psi \vee \psi \mid \psi \wedge \psi \mid \neg\psi \\ t & ::= & x \mid c \mid f(\bar{t}) \mid \mathsf{ite}(\psi, t, t') \end{array}$$

Fig. 7. Syntax for FO-LFP, where φ is the starting nonterminal, P ranges over definable relation symbols, and R and f range over relations and functions from a signature.

6.2 Separator Realizability and Synthesis with Least Fixed Points

We now develop a solution for FO-LFP(k, k') synthesis, where FO-LFP(k, k') is the logic FO-LFP restricted to k first-order variables and k' definable relation symbols $\{P_1, \dots, P_{k'}\}$, which we assume are disjoint from the relation symbols in the signature τ. We extend the ranked alphabet for FO(k) by representing definitions with binary symbols "let $P(\bar{x})$" whose children are the definition body and the remainder of the formula that uses the definition (superscripts denote arity):

$$\Sigma_{\text{FO-LFP}(k,k')} = \left\{ \mathsf{let}\, P(\bar{x})^2,\ P^{\mathsf{arity}(P)} \;\middle|\; P \in \{P_1, \dots, P_{k'}\},\ \bar{x} \in V^{\mathsf{arity}(P)} \right\} \cup \Sigma_{\text{FO}(k)}$$

In this problem we use *two-way tree automata*, which can navigate an input tree in both directions (from a node to its children or to its parent), thus making all parts of a tree accessible from any node. When reading an occurrence of a definable relation symbol P, the automaton can navigate to the corresponding definition, which is elsewhere in the tree, and read it. This same capacity to move up and down in the tree gives us an elegant way to describe the evaluation of *recursive definitions*, which must be read multiple times to compute.

6.2.1 Two-Way Tree Automata. A two-way tree automaton $\mathcal{A} = \langle Q, \Sigma, I, \delta, F \rangle$ on κ-ary trees generalizes the transition function of ATAs to have the form $\delta : Q \times \Sigma \to \mathcal{B}^+(Q \times \{-1, \dots, \kappa\})$, where occurrences of -1 in a transition require the automaton to *ascend* in the input tree. Runs are defined as for ATAs, with the exception that a run cannot ascend above the root of the input tree. For our purposes, we modify the notion of acceptance by distinguishing a set of final states $F \subseteq Q$. A tree is accepted by a two-way automaton if there is a run for which every branch reaches a state in F. Note that the two-way tree automata we use here are no more expressive than alternating tree automata, and there are algorithms to convert a two-way automaton to a one-way nondeterministic automaton in exponential time [Vardi 1998], and thus emptiness is decidable.

6.2.2 Automaton for Evaluating Formulas with Recursive Definitions. For simplicity, we again describe a construction for the simpler variant of FO-LFP(k, k') without functions and if-then-else terms. The ranked alphabet for this simplification looks as follows:

$$\Sigma'_{\text{FO-LFP}(k,k')} = \left\{ \mathsf{let}\, P(\bar{x})^2,\ P(\bar{x})^0 \;\middle|\; P \in \{P_1, \dots, P_{k'}\}, \bar{x} \in V^{\mathsf{arity}(P)} \right\} \cup \Sigma'_{\text{FO}(k)}$$

Let us assume each symbol P_i has $\mathsf{arity}(P_i) = r$. For a fixed structure A with $|\mathsf{dom}(A)| = n$, we define a two-way tree automaton $\mathcal{A}_A = \langle Q, \Sigma'_{\text{FO-LFP}(k,k')}, I, \delta, F \rangle$ whose language is the set of sentences in (simplified) FO-LFP(k, k') that are true in A. We discuss each component next.

States. In addition to assignments, each state keeps track of information that enables the automaton to evaluate recursively-defined relations. This includes (1) whether the automaton is going *up* to find a definition or *down* to evaluate a formula, (2) a counter value from $Count \triangleq \{0, \dots, n^r\}$ that tracks the stage of the current least fixed point computation, and (3) the current definition being evaluated (if any), i.e., a member of the set $Defn \triangleq \{\bot, P_1, \dots, P_{k'}\}$. Finally, some states have a tuple of domain elements from $Val \triangleq \mathsf{dom}(A)^r$ rather than a partial assignment, which is used to pass values to the body of a definition whenever a defined relation is used. Note that the distinction between a tuple and an assignment takes care of part (1) above. Similar to partial assignments, the tuples are marked to indicate the automaton's mode of operation: checking a formula is either true (verifying) or false (falsifying). Combining the above, we have

$$Q \coloneqq \mathit{Dual}(\mathit{Assign}) \times \mathit{Count} \times \mathit{Defn} \ \cup \ \mathit{Dual}(\mathit{Val}) \times \mathit{Count} \times \mathit{Defn} \ \cup \ \{q_f\},$$

where q_f is distinct from all other states and $F = \{q_f\}$.

The states Q can be divided into two categories: *up* and *down*. The *up* states correspond to checking membership in a defined relation. In an *up* state $\langle \mathit{val}, \mathit{count}, \mathit{defn}\rangle \in \mathit{Dual}(\mathit{Val}) \times \mathit{Count} \times \mathit{Defn} \subseteq Q$, the automaton navigates up on the input tree to find the definition for *defn*, and it carries a tuple *val* of domain elements that it should check for membership in the defined relation. In a *down* state $\langle \mathit{assign}, \mathit{count}, \mathit{defn}\rangle \in \mathit{Dual}(\mathit{Assign}) \times \mathit{Count} \times \mathit{Defn} \subseteq Q$, the automaton evaluates a formula under the variable assignment *assign* while navigating down in the input tree.

Initial states. There is one initial state containing an empty assignment, a counter at 0, and the current definition set to $\perp$, i.e., $I = \{\langle \emptyset, 0, \perp\rangle\}$.

Transitions. The transitions for symbols shared with FO are similar to the earlier construction (§5.1.1). The novelty is to define transitions for definitions and occurrences of defined relations $P_i(\bar{x})$. For intuition, consider the increasing sequence of "approximations" for a relation defined by a formula ψ. The least fixed point for the operator F_ψ (see §6.1) can be computed in n^r steps by iteratively applying F_ψ starting from $\emptyset$, giving us the sequence

$$\emptyset \subseteq F_\psi(\emptyset) \subseteq \cdots \subseteq F_\psi^i(\emptyset) = F_\psi^{i+1}(\emptyset),$$

where $i \leq n^r$ (follows from monotonicity of F_ψ). When the automaton reads a defined relation $P_i(\bar{x})$ in a state $\langle \gamma, j, P_i\rangle$, it will attempt to verify that $\gamma(\bar{x}) = \bar{a} \in F_\psi^j(\emptyset)$. Similarly, in a state $\langle \tilde{\gamma}, j, P_i\rangle$ it will attempt to verify that $\gamma(\bar{x}) = \bar{a} \notin F_\psi^j(\emptyset)$.

Presenting all of the many transitions would obscure the main ideas, so we give only a description of interesting ones here; a full account can be found in [Krogmeier and Madhusudan 2021]. We focus on four cases: (1) reading a defined symbol, (2) finding a definition, (3) reading a definition, and (4) a variation on (1) where the defined symbol being read is not the current definition. Below, we use $\mathit{assign} \in \mathit{Dual}(\mathit{Assign}), \gamma \in \mathit{Assign}, \mathit{val} \in \mathit{Dual}(\mathit{Val}), v \in \mathit{Val}, j \in \mathit{Count}$, and $P_i, P_j \in \mathit{Defn}$.

(1) *Reading a defined symbol.* Suppose the automaton reads "$P_i(\bar{x})$" in a *down* state $\langle \mathit{assign}, j, P_i\rangle$. Thus it is currently reading the definition for P_i (call the definition ψ) and has encountered a *use* of P_i in the form $P_i(\bar{x})$. Suppose $j = 0$. If $\mathit{assign} = \gamma$, then the automaton is *verifying* and it must verify that $v = \gamma(\bar{x})$ is in the jth stage of the least fixed point computation for the definition of P_i. The transition for this case is False, since $v \notin \emptyset = F_\psi^0(\emptyset)$. Otherwise, if $\mathit{assign} = \tilde{\gamma}$, then the automaton is *falsifying* and must verify that $v = \gamma(\bar{x})$ is *not* in the jth stage. So the transition for this case is True, since $v \notin \emptyset$. If $j > 0$, in both cases the transition forces the automaton to navigate up to the definition and evaluate it. It does this by changing to the *up* state $\langle v, j, P_i\rangle$, if verifying, and to the up state $\langle \tilde{v}, j, P_i\rangle$, if falsifying.

(2) *Finding a definition.* Suppose step (1) has just occurred, and thus the automaton is looking for a definition of P_i. The automaton continues moving up on the input tree until it encounters a symbol that marks the definition of P_i, i.e., a symbol of the form "let $P_i(\bar{x})$". Note: if the definition does not exist in the tree, then the automaton continues to the root, at which point it can make no valid transition and the tree is rejected.

(3) *Reading a definition.* Suppose step (2) has just occurred and the automaton is in state $\langle val, j, P_i \rangle$ reading "let $P_i(\bar{x})$". The automaton decrements the counter and proceeds to evaluate the definition by moving into the child tree corresponding to the definition of P_i. It enters a *down* state with assignment $\gamma = \cup_i \{x_i \mapsto v_i\}$ that maps variables in $\bar{x}$ to the passed values in *val*. If the automaton is verifying ($val = v$), then the new state is $\langle \gamma, j-1, P_i \rangle$. Otherwise, the automaton is falsifying ($val = \tilde{v}$) and the new state is $\langle \tilde{\gamma}, j-1, P_i \rangle$.

(4) *Reading a new defined symbol.* Suppose the automaton is reading "$P_i(\bar{x})$" in a *down* state $\langle assign, count, P_j \rangle$, where $P_j \neq P_i$. Thus it has encountered a *use* of a defined relation P_i that it is not currently reading. Rather than checking the value of *count*, as in case (1) above, the automaton continues on to case (2) with the current definition set to P_i and with a fresh counter initialized at $count = n^r$.

Acceptance. Recall that the automaton accepts a tree t if it has a run on t where every branch reaches the final state q_f. We note that the full set of transitions ensures that the only way to reach the final state q_f is via True transitions.

Lemma 4. $\mathcal{A}_A$ accepts any sentence $\varphi \in \Sigma'_{\mathsf{FO\text{-}LFP}(k,k')}$ that is true in A.

6.2.3 Decision Procedure. We define automata $\mathcal{A}_A$ for each input structure A as described in the construction, with the proviso that negative structures have initial states $I = \{\langle \tilde{\varnothing}, 0, \bot \rangle\}$. We take the product of these automata as before and convert the resulting automaton to a one-way nondeterministic automaton without alternation by adapting the technique of [Vardi 1998]. We further take the product of the nondeterministic automaton with the grammar automaton $\mathcal{A}_G$. Checking emptiness of the final automaton, which has size $O(2^{poly(mn^k k')}|G|)$, gives us the decision procedure for realizability and synthesis. Again, it is straightforward to adapt the construction to full $\mathsf{FO\text{-}LFP}(k,k')$ with if-then-else and function terms, giving us:

Theorem 5. $\mathsf{FO\text{-}LFP}(k,k')$-separator realizability and synthesis *is decidable in* EXPTIME *for a fixed signature and fixed* $k, k' \in \mathbb{N}$.

7 TERM SYNTHESIS

In this section we show that it is possible to use the same general approach to *synthesize terms.* We examine the term synthesis problem for a logic similar to FO-LFP with recursively-defined functions (adaptations for other logics are similar). Much of the construction is similar to the construction for FO-LFP (§6). We give the main idea by showing how the evaluation automaton's state space changes, and we describe at a high level some new transitions related to terms.

7.1 First-Order Logic with Least Fixed Points and Recursive Functions

We consider term synthesis for a logic with least fixed point relations and *recursively-defined functions.* Recall the list merge example from Figure 4. Given a set of (unlabeled) structures that each interpret the constants $in_1, \ldots, in_d$, and *out* as an input-output example for some functional relationship (e.g., $merge(in_1, in_2) = out$), the goal is to decide whether there is a term t that evaluates to *out* in each structure, and to synthesize one if it exists. We explore this problem for the language FO-LFP-FUN (syntax in Figure 8), which is similar to FO-LFP and, additionally, has recursively-definable functions and yields *terms* rather than formulas.

The semantics for FO-LFP-FUN coincides with FO-LFP on shared features. The only novelty is in how we define the semantics of recursive functions and, in particular, how we account for functions that may not be total on the domain of the structure A. We choose to interpret them as

$$
\begin{array}{lllllll}
\varphi & ::= & \text{let } g(\bar{x}) =_{\mathsf{lfp}} t \text{ in } \varphi & \qquad \psi & ::= & \exists x.\psi \mid \forall x.\psi \mid R(\bar{t}) \mid P(\bar{t}) \\
& \mid & \text{let } P(\bar{x}) =_{\mathsf{lfp}} \psi \text{ in } \varphi & & & \mid \psi \vee \psi \mid \psi \wedge \psi \mid \neg\psi \\
& \mid & t & \qquad t & ::= & x \mid c \mid f(\bar{t}) \mid \mathsf{ite}(\psi, t, t')
\end{array}
$$

Fig. 8. Syntax for FO-LFP-FUN with φ the starting nonterminal. The language is similar to FO-LFP, but it yields terms rather than formulas and adds (recursively) definable functions g.

partial functions on $\mathsf{dom}(A)$ and to interpret formulas in a 3-valued logic. Note that in this setting it is also simple and convenient to allow input structures to interpret function symbols from the signature as partial functions. For instance, in the *merge* example from Figure 4, we might prefer to specify *head* as a partial function that is only defined on elements that denote lists. We give here a high-level description of the semantics for recursive functions; details can be found in [Krogmeier and Madhusudan 2021].

Semantics for Recursive Functions. A defined function g with $\mathsf{arity}(g) = d$ is interpreted as a partial function $g^A : \mathsf{dom}(A)^d \rightharpoonup \mathsf{dom}(A)$, which is a member of the bottomed partial order $O = \langle \mathsf{dom}(A)^d \rightharpoonup \mathsf{dom}(a), \sqsubseteq, \perp \rangle$, where $\perp$ is undefined everywhere and $f \sqsubseteq f'$ holds if for all $\bar{a} \in \mathsf{dom}(A)^d$, whenever $f(\bar{a})$ is defined, then $f'(\bar{a})$ is defined and $f(\bar{a}) = f'(\bar{a})$. This partial order has finite height since all structures here are finite. Suppose g is defined recursively using a term $t(x_1, \ldots, x_d, g)$. We associate a monotone function $F_t : O \rightarrow O$ to the defining term t, and let g^A be the least fixed point of F_t, which can easily be shown to exist and to be equal to the stable point of the chain $\perp \sqsubseteq F_t(\perp) \sqsubseteq \cdots \sqsubseteq F_t^i(\perp) = F_t^{i+1}(\perp)$, with $i \leq |\mathsf{dom}(A)^d|$. The syntax and semantics for terms in FO-LFP-FUN guarantee monotonicity of the function F_t for any term t. It follows that least fixed points exist for each definition. In general, however, care is needed to ensure that the least fixed point is total on $\mathsf{dom}(A)^d$, and whether or not this is so depends on the definition and the structure A.

7.2 Automaton for Evaluating Terms with Recursive Functions

We now sketch the main idea for defining an automaton that accepts all terms which evaluate to a given domain element a in a structure A. We consider the language FO-LFP-FUN(k, k'), which restricts FO-LFP-FUN to k first-order variables and k' definable relations and functions from $P = \{P_1, \ldots, P_{k_1}\}$ and $F = \{g_1, \ldots, g_{k_2}\}$, respectively, with $k' = k_1 + k_2$. A ranked alphabet for $\Sigma_{\text{FO-LFP-FUN}(k,k')}$ extends $\Sigma_{\text{FO-LFP}(k,k')}$ in an obvious way. Note that here we consider input trees representing arbitrarily deep logical terms, in contrast to our earlier simplifications.

Fix a structure A of size $n = |\mathsf{dom}(A)|$ and fix a domain element $a \in \mathsf{dom}(A)$. We want to define a two-way automaton $\mathcal{A}_A = \langle Q_{\text{TERM}}, \Sigma_{\text{FO-LFP-FUN}(k,k')}, I, \delta, F \rangle$ that accepts the set of closed terms $t \in$ FO-LFP-FUN(k, k') such that $t^A = a$. For simplicity, assume r-ary functions and relations only. The states are as follows:

$$
\begin{aligned}
Q_{\text{TERM}} &\coloneqq \mathsf{EvalForm} \cup \mathsf{EvalTerm} \cup \{q_f\} \\
\mathsf{EvalForm} &\coloneqq (\mathit{Dual}(\mathit{Assign}) \cup \mathit{Dual}(\mathit{Val})) \times \mathit{Count} \times \mathit{Defn} \\
\mathsf{EvalTerm} &\coloneqq (\mathit{Assign} \cup \mathit{Val}) \times \mathit{Count} \times \mathit{Defn} \times \mathsf{dom}(A)
\end{aligned}
$$

The sets $\mathit{Assign} \triangleq V \rightharpoonup \mathsf{dom}(A)$, $\mathit{Val} \triangleq \mathsf{dom}(A)^r$, $\mathit{Dual}(X) \triangleq \{x, \tilde{x} \mid x \in X\}$, $\mathit{Count} \triangleq \{0, \ldots, n^r\}$, and $\mathit{Defn} \triangleq \{\perp, P_1, \ldots, g_{k_2}\}$ serve the same purposes as before. Notice that the automaton has a new category of states, namely, those for evaluating terms. The transitions related to formulas are very similar to those for FO-LFP. Below, we give three representative cases for the transition function δ. Let $\gamma \in \mathit{Assign}$, $j \in \mathit{Count}$, and $g, \mathit{defn} \in \mathit{Defn}$.

Reading variables. The automaton is reading "x" and verifying that the input tree evaluates to $a \in \mathrm{dom}(A)$. It only needs to check that the variable x is mapped to a in the current assignment γ:

$$\delta(\langle \gamma, j, \mathit{defn}, a\rangle, x) \;=\; \begin{cases} \mathsf{True} & \gamma(x)\downarrow,\ \gamma(x) = a \\ \mathsf{False} & \text{otherwise} \end{cases}$$

Reading if-then-else terms. The automaton is reading "ite" and verifying that the input tree evaluates to $a \in \mathrm{dom}(A)$. It must either (1) verify the condition formula (first child) *and* verify that the term in the "then" branch (second child) evaluates to a or (2) falsify the condition formula *and* verify that the term in the "else" branch (third child) evaluates to a:

$$\delta(\langle \gamma, j, \mathit{defn}, a\rangle, \mathrm{ite}) \;=\; (\langle \gamma, j, \mathit{defn}\rangle, 1) \wedge (\langle \gamma, j, \mathit{defn}, a\rangle, 2) \vee (\langle \tilde{\gamma}, j, \mathit{defn}\rangle, 1) \wedge (\langle \gamma, j, \mathit{defn}, a\rangle, 3)$$

Reading a defined function g. The automaton is reading "g" and the current definition is set to g', with $g' \neq g$, analogous to case (1) from §6.2.2. The automaton "guesses and checks" that the argument terms for g evaluate to $\bar{a}$ and ascends to the definition of g with a fresh counter set to n^r to verify that g evaluates to a when applied to $\bar{a}$:

$$\delta(\langle \gamma, j, g', a\rangle, g) \quad = \bigvee_{\bar{a} \in \mathit{Val}} \left((\langle \bar{a}, n^r, g, a\rangle, -1) \wedge \bigwedge_{i \in [r]} (\langle \gamma, j, g', a_i\rangle, i) \right) \qquad (g' \neq g)$$

The rest of the transitions follow along these lines and are similar to those for FO-LFP. There is a single initial state with $I = \{\langle \emptyset, 0, \bot, a\rangle\}$, and the acceptance condition is again reachability with $F = \{q_f\}$. Similar reasoning to that for the FO-LFP construction can be used to show:

Theorem 6. FO-LFP-FUN(k, k')-term synthesis *is decidable in* EXPTIME *for a fixed signature* τ *and fixed* $k, k' \in \mathbb{N}$.

The idea sketched here can be easily added to earlier constructions without increasing complexity in order to solve synthesis for logics with the full gamut of terms and, in particular, to solve term synthesis for FO(k) in the same complexity as the separability problem.

8 LOWER BOUNDS

Here we present lower bounds arguing the upper bound complexity we obtain on certain parameters of the problem is indeed tight. Given the number of logics and variants (and problems for separators, queries, and terms), we focus on lower bounds only for FO(k); of course, these also give lower bounds for more expressive languages and variants.

8.1 A Lower Bound for Separability in FO(k)

The upper bound for separability in FO(k) from §5 is linear in the size of the grammar and exponential in mn^k, where m is the number of input structures and n is the maximum size of any input structure. Hence, for a fixed k, the algorithm we propose is exponential time in the size of the input. We show a matching lower bound (this can be adapted for queries and terms as well).

Theorem 7. FO(k)-separator realizability *is* EXPTIME*-hard for any fixed* $k > 4$.

Proof. The reduction is from the word acceptance problem for an alternating polynomial space Turing machine. Given an alternating Turing machine M and an input w, with M using space s that is polynomial in $|w|$, the reduction yields s positively labeled first-order τ-structures $A_1, \ldots, A_s$, with $|\mathrm{dom}(A_i)| = O(s)$, and a grammar G of size polynomial in $|\langle M, w\rangle|$. The signature τ depends only on M. Each structure consists of two parts: (1) a cycle of length s and (2) a gadget encoding the transition relation for M along with unique constants for tape symbols from the machine's tape alphabet Γ. (We use constants that encode the machine head and state, i.e., $\Gamma' = \Gamma \times Q \cup \Gamma$.)

Let us consider the purposes of the *structures* and the *grammar*, which are complementary. The structures can be viewed as distinct copies of the machine M that are used to verify that a computation tree for M on w, encoded in a sentence from the grammar, obeys the transition relation for each of the s tape cells. The grammar can be viewed as a skeleton of *computation trees* for M on input w. A particular sentence φ from the grammar G encodes a computation tree of potentially exponential depth, and it asserts many things about the structure on which it is interpreted. When understood together across all structures, the truth of the assertions is equivalent to the computation tree being an *accepting computation tree*, i.e., that successive configurations follow the transition relation and the final configuration is accepting. The main trick is to emulate in the grammar the generation of each successive machine configuration in a way that allows checking the transition relation. This is not entirely straightforward because s tape symbols cannot all be stored at once in k variables (k is fixed). Here is some intuition for how we accomplish this.

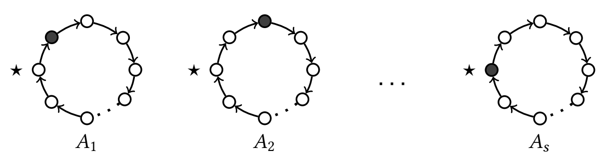


Fig. 9. Structure A_i tracks the window centered on the ith cell and has a special cycle element (the dark node) of distance i from the start of the cycle, denoted $\star$. As the symbols of a configuration are produced, the variables $\bar{y}$ are equated with the current tape window if the cycle pointer is equal to the dark element.

Each of the s structures is made to track a distinct window of three contiguous tape cells (as well as the previous contents for the window). The grammar uses a polynomial-sized gadget of nonterminals to iteratively produce the tape cell contents of a given configuration. Refer to Figure 9 in the following for a picture of how this gadget works over each structure. In each iteration, the grammar moves a *ptr* variable along the cycle of size s. If *ptr* is equal to a special element, filled dark in Figure 9, then the grammar requires the current tape cell's contents (and neighbors) to be stored in variables by asserting an equality. Each structure is made to track a unique window by differently interpreting the distance between a starting node, denoted $\star$, and the special dark node. The grammar G ensures the following invariant holds for all sentences $\varphi \in L(G)$: if we evaluate φ in A_i, then upon evaluating the subformula of φ that picks a symbol for cell $i+1$, window i of the previous configuration is stored in variables x_1, x_2, x_3 and window i of the current configuration is stored in variables y_1, y_2, y_3. The grammar checks that successive windows obey the transition relation by asserting the relation $\delta(\mathit{choice}, \bar{x}, \bar{y})$, where δ encodes the transition relation for M and $\mathit{choice} \in \{0, 1\}$ encodes which of two transitions for the alternating machine is being verified.

More details can be found in [Krogmeier and Madhusudan 2021]. □

Theorem 8. $\mathsf{FO}(k)$-query realizability *is* EXPTIME-*hard for fixed* $k > 4$.

Proof. Reduction from $\mathsf{FO}(k)$-*separator realizability*. Positive structures have a full query answer set and negative structures have an empty query answer set. □

Theorem 9. $\mathsf{FO\text{-}LFP}(k, k')$-separator realizability *is* EXPTIME-*hard for fixed* $k, k' \in \mathbb{N}$ *with* $k > 4$.

Proof. Reduction from $\mathsf{FO}(k)$-*separator realizability*. □

8.2 More Lower Bounds and Open Problems

We can now ask whether separator realizability for $\mathsf{FO}(k)$ is decidable in polynomial time if there is only *one* structure. With only one structure (positively labeled, say) the problem of separability may seem odd, but checking whether there is *any* sentence in the grammar G that is true on the single structure is actually a nontrivial problem; indeed, the grammar is quite powerful.

We can show a general reduction from separator realizability for multiple structures to realizability for a single structure (but over a different grammar and for $k' = k + 1$). Given a set of positive structures *Pos*, negative structures *Neg*, and a grammar G (over $\Sigma_{\mathsf{FO}(k)}$), we can reduce the realizability problem to a new realizability problem over a single positive structure M and a grammar G' (over $\Sigma_{\mathsf{FO}(k+1)}$). The idea is that M has (a) copies of all the structures in *Pos* and *Neg*, (b) a set of elements i, one for each $i \in |Pos \uplus Neg|$, which represent *structure identifiers*, (c) unary relations *Id* and *P*, where *Id* holds for the set of structure identifiers i and P holds for the set of identifiers for structures in *Pos*, and (d) a binary relation *Owns* that associates each i with the elements of the copy of structure i in M. The grammar G' is designed to generate only formulas of the form $\forall i.Id(i) \rightarrow (P(i) \leftrightarrow \alpha'(i))$. The formula α' is obtained by taking a formula α admitted by G and relativizing the quantification so that it is restricted to those elements that are associated to i. For example, $\forall x.\beta(x)$ is relativized to $\forall x.Owns(i, x) \rightarrow \beta(x)$ and $\exists x.\beta(x)$ is relativized to $\exists x.Owns(i, x) \wedge \beta(x)$. A formula $\varphi' \in L(G')$ is true in M if and only if there is a formula $\varphi \in L(G)$ that is a separator for *Pos* and *Neg*. Note that the formulas in G' use one extra variable (namely i).

This reduction combined with Theorem 7 shows the following:

Theorem 10. *For any fixed $k > 5$, given a single structure M and a RTG G over $\mathsf{FO}(k)$, checking whether there is a formula in $L(G)$ that is true in M is* EXPTIME*-complete.*

Open Problems. Our algorithm has exponential dependence on the number of structures m. We do not know whether algorithms polynomial in m can be achieved. More precisely, we do not know if separator realizability can be achieved in time $O(f(m, n, k) \cdot g(n, k))$, where f is a polynomial function and g is an arbitrary function. Learning algorithms that scale linearly or polynomially with the number of data samples are clearly desirable.

Interestingly, if there is no grammar restriction, i.e., we look for a separator in $\mathsf{FO}(k)$, then such an algorithm is indeed possible. This follows from a suggestion by Victor Vianu [Vianu 2020]. The algorithm works on the basis of $\mathsf{FO}(k)$-types [Libkin 2004], which capture equivalence classes of finite structures that cannot be distinguished from each other by any $\mathsf{FO}(k)$ formula. Of crucial importance is the fact that these equivalence classes of structures can be *defined* by an $\mathsf{FO}(k)$ formula, which can be effectively computed for a given structure. Consequently, we can *independently* compute the defining formula, denoted $type(A_i)$, for each $A_i \in Pos$ and then form the disjunction $\psi := \bigvee_i type(A_i)$. If ψ holds for any structure in *Neg* then there can be no separator. Otherwise ψ is a separator. This procedure works in time polynomial in the number of structures.

However, we do not see any way to adapt the above procedure to arbitrary grammars. Furthermore, it has a disadvantage as an algorithm for learning— it yields very large formulas that essentially overfit the positive samples. In contrast, the automata-theoretic method can find the smallest formulas.

There are, of course, many lower bound problems that are open for different logics and variants, and each of them has many parameters ($|G|$, m, n, k, k', k_1, k_2, as well as the arities of symbols). One can ask several parameterized complexity [Flum and Grohe 2006] lower bound questions for each of our problems, and we leave this to future work. In particular, one key question involves the parameter k (which we have assumed is fixed in most of our treatment): is the double exponential dependence on k tight?

9 FURTHER RESULTS AND DISCUSSION

In this section we discuss how the technique illustrated in §5, §6, and §7, can be adapted to solve problems in two other settings: logic programming and second- and higher-order logics. We also remark on the generality of the approach and give a connection to Ehrenfeucht-Fraïssé games.

9.1 Mutual Recursion and Logic Programming

Recall that our treatment of FO-LFP from §6 did not include *mutually-recursive* definitions. In fact, mutual recursion can be handled with a modest increase in the number of automaton states. Consider a variant of FO-LFP that allows *blocks* of defined relations, in which all relations in a single block can refer to each other in their definitions, like the following:

$$\mathsf{let}\ \left\{\begin{array}{lll} P_1(x_1,x_2) & =_{\mathsf{lfp}} & \varphi_1(x_1,x_2,P_1,P_2) \\ P_2(x_1,x_2) & =_{\mathsf{lfp}} & \varphi_2(x_1,x_2,P_1,P_2) \end{array}\right\}\ \mathsf{in}\ \varphi(P_1,P_2)$$

The semantics for blocks of mutually-recursive definitions can be defined in terms of a *simultaneous fixed point*. For the example above, we can define functions $F_1, F_2 : 2^X \times 2^X \to 2^X$, where $X = \mathsf{dom}(A)^2$, and for $X_1, X_2 \subseteq 2^X$:

$$F_i(X_1,X_2) \triangleq \left\{ \bar{a} \in X \;\middle|\; A \models \varphi_i(\bar{a}/\bar{x}, X_1/P_1, X_2/P_2) \right\} \quad i \in \{1,2\}$$

We can interpret the relations P_1 and P_2 as the components of the simultaneous least fixed point of the system of equations above; see [Fritz 2002] for more on simultaneous fixed points.

Evaluating Mutually-Recursive Definitions. In the spirit of our technique, we ask how an automaton can check membership for a relation defined by mutual recursion using state bounded by the structure. The same ideas carry over from §6 with a modification. As before, all tuples can be associated with the stage at which they enter the (now) *simultaneous* fixed point computation, and the automaton can use counters to check membership at a given stage. However, the number of stages grows exponentially in the number of relations in a block of definitions (which we can assume is bounded by the number of definable symbols k'). The automaton state must now include a *product* of counters, one for each definable symbol. Other than this change to the states, the construction that handles mutual recursion in FO-LFP remains essentially the same.

Logic Programming. With mutually-recursive definitions, our technique can be used to solve Datalog synthesis problems; this is not surprising since Datalog is logically similar to standard first-order logics with least fixed points (in fact, it corresponds to an existential fragment ∃LFP of first-order logic with least fixed points that only allows negation on atomic relations from the signature and disallows universal quantification [Libkin 2004]). See [Krogmeier and Madhusudan 2021] for more details on a Datalog synthesis problem that our technique can solve. (We note for a fixed number of variables and definable relations, the space of Datalog programs is finite and thus decidability is not theoretically interesting.) We can also model problems from *inductive logic programming* (ILP) [Muggleton and de Raedt 1994], e.g., learning from entailment over bounded variable horn-clause programs, by encoding background knowledge (a set of definite horn clauses) in the grammar.

9.2 Second-Order Logic

The approach naturally extends to second-order logic (SO) (see, e.g., [Libkin 2004] for syntax and semantics). We state here a result for relational $\mathsf{SO}(k,k')$, a version of SO restricted to k first-order variables and k' second-order relation variables. An alternating one-way automaton $\mathcal{A}_A$ can evaluate $\mathsf{SO}(k,k')$ formulas on a fixed structure A by keeping track of an assignment to k

first-order variables and an assignment to k' second-order relation variables of maximum arity r (the relation variables map to sets of r-tuples) using a state space of size $O(2^{(k'n^r)}n^k)$. The decision procedure follows the same lines as before.

Theorem 11. $\mathsf{SO}(k, k')$-separator realizability and synthesis *is decidable in* 2EXPTIME *for a fixed signature and fixed* $k, k' \in \mathbb{N}$.

The same idea sketched above easily extends to logics with variables over higher-order functions.

9.3 Discussion

We believe the tree automata-theoretic approach proposed in this work is extremely versatile. The crux is to build automata that, when reading the parse tree of an expression, can evaluate it on a fixed structure using finitely many states. This typically is true if there is a way to recursively evaluate the semantics of expressions using memory that depends on the size of the structure but *not on the size of expressions*. Bounding the number of variables is one way to achieve this.

We claim our technique applies to any logic or language for which (a) the semantics of expressions can be described locally in the parse tree in terms of the semantics for subexpressions and (b) evaluating the semantics at each node of the parse tree requires memory that is bounded by a function of the structure size (and not the formula size). The fact that logics with definitions and recursion can be captured with two-way tree automata shows that they also meet these conditions, since the automaton can be converted to a deterministic bottom-up automaton. We leave formalizing this claim, proving it, and finding further instantiations of the technique to future work.

Finally, we note that the separability problems considered here can be viewed from the perspective of Ehrenfeucht-Fraïssé games [Ehrenfeucht 1961; Fraïssé 1953], which are typically used to show formulas in a logic can or cannot distinguish between two structures. The separability problem instead asks whether a set of positive structures can be separated from a set of negative structures using formulas that conform to a given grammar. Hence the game in our setting is one that is specific to the given grammar and furthermore forces the players to play *simultaneously* on all the structures. We leave further investigation of this relationship to future work.

10 RELATED WORK

Program Synthesis from Examples. Learning logical formulas is closely related to *program synthesis*, and especially, program synthesis from *examples* (as opposed to deductive approaches from specifications [Manna and Waldinger 1980]). Synthesis from examples, or *programming by examples* (PBE), has been active in recent years and has seen successes in practice (e.g., [Polozov and Gulwani 2015]). In PBE, the goal is to synthesize a program consistent with a set of input-output examples; several domains have been explored, e.g., synthesis of database queries from examples [Shen et al. 2014; Thakkar et al. 2021; Wang et al. 2017a] and from analysis of database-backed application code [Cheung et al. 2013], synthesis of data completion scripts [Wang et al. 2017c], data structure transformations [Feser et al. 2015], and typed functional programs [Osera and Zdancewic 2015; Polikarpova et al. 2016]. A common approach to PBE involves *version space algebra* [Mitchell 1982], where the idea is to capture the set of all programs that work on each example in a compact representation and then intersect the sets for each example to represent programs consistent with all examples (e.g., see [Gulwani 2011]). Our approach essentially uses tree automata as a version space algebra to capture *all* logical expressions that satisfy some criterion over input structures.

Synthesis with Grammar. Using grammar to constrain the hypothesis space follows a line of work in program synthesis that uses syntactic biases like partial programs, e.g. [Solar-Lezama et al. 2006], and more broadly, *syntax-guided synthesis* (SyGuS) for logics (typically logics supported by SMT theories) [Alur et al. 2015]. In a SyGuS problem, one is given a grammar from which to synthesize a logical expression (similar to the setting in this paper) as well as a specification in the form of a universally-quantified formula that refers to a placeholder e, which must be valid when the synthesized expression is plugged in for e. The separability problem can in fact be formulated as a SyGuS problem, though SyGuS divisions and tools only support synthesis of *quantifier-free formulas*. There is a large body of work exploring program synthesis and syntax-guided synthesis for quantifier-free logics that focuses on practical and scalable techniques, and for the most part does not offer any guarantee of completeness. When grammars admit infinitely many expressions, these solvers cannot report *unrealizability*, which is in general undecidable [Caulfield et al. 2015]. The ability to decide realizability and synthesis is a crucial difference in our work.

Decidable Realizability and Synthesis. For systems and programs that have *finite state spaces*, the realizability problem has been extremely well studied and a rich class of specifications for such systems is known to admit decidable realizability. The crux of the techniques used in this domain rely on tree automata that work on *infinite* trees and infinite games played on finite graphs (while our work uses tree automata on finite trees). This problem was first proposed by Church [Church 1960], and a rich theory of realizability/synthesis has emerged [Buchi and Landweber 1969; Grädel et al. 2002; Kupferman et al. 2000, 2010; Madhusudan and Thiagarajan 2001; Pnueli and Rosner 1989, 1990; Rabin 1972]. The key idea is to encode the branching behavior of a reactive system using an infinite tree and build automata that accept systems (trees) whose behaviors satisfy a specification.

Our work is technically closer to the approach in [Madhusudan 2011], which studies synthesizing *imperative reactive programs* over a finite number of variables ranging over finite domains with logical specifications (e.g., linear temporal logic). The decidability of realizability/synthesis is proved using tree automata that work on finite trees (parse trees of programs), similar to the work presented here. Unlike our work, the tree automata have infinitary acceptance conditions in order to capture properties of infinite executions of programs. Other differences include (1) our work interprets logical expressions over *unbounded* structures, and (2) the specification for synthesis is not a logical formula, but rather a set of labeled structures. Intuitively, we trade the power of logical specifications in [Madhusudan 2011] and replace it with a finite set of structures in order to synthesize over unbounded domains. Though the constructions in our work are too large to implement naïvely, the core idea to use tree automata on parse trees of expressions for synthesis has been made practical in some recent work, e.g., for string and matrix transformations [Wang et al. 2017b] and string encoders/decoders and comparators [Wang et al. 2018].

Decidability results for synthesis of expressions over unbounded data domains are uncommon, though there are some recent results for restricted classes of programs and models of computation, e.g., synthesizing finite-state transducers [Khalimov et al. 2018] and synthesizing a restricted class of imperative programs [Krogmeier et al. 2020]. In [Krogmeier et al. 2020], the authors study the problem of synthesizing uninterpreted imperative programs from a given grammar, where programs come with assertions that must be satisfied for any interpretation of function and relation symbols over any domain, possibly infinite. For the restricted subclass of coherent programs [Mathur et al. 2019], there is a decision procedure based on tree automaton emptiness, and, similar to our work, the solution uses tree automata working over parse trees. There is also recent work giving sound techniques for proving unrealizability of SyGuS problems [Hu et al. 2019], and, more recently, a decision procedure for SyGuS problems over linear integer arithmetic with conditionals over finitely-many examples [Hu et al. 2020].

Learning Logical Formulas. In [Koenig et al. 2020], the authors study a separability problem for first-order logic formulas with *bounded quantifier depth*. In contrast to the problems we consider in this work, bounding the quantifier depth makes the search space finite up to logical equivalence, enabling a reduction to and from SAT. There is also work on the decidability of learning separators from labeled examples for various description logics [Funk et al. 2019; Jung et al. 2020]. There, separation problems are studied in the presence of an ontology, which is a finite set of logical sentences. The presence of ontologies makes the problem different from our work; adapting our general synthesis approach to the world of description logics remains future work. There is also prior work studying the complexity of learning logical concepts by characterizing the VC-dimension of logical hypothesis classes [Grohe and Turán 2004], work on parameterized complexity for logical separation problems in the PAC model [van Bergerem et al. 2021], learning MSO-definable concepts on strings [Grohe et al. 2017] and concepts definable in first-order logic with counting [van Bergerem 2019], learning temporal logic formulas from examples [Neider and Gavran 2018], and learning quantified invariants for arrays [Garg et al. 2015].

Inductive Logic Programming. In *inductive logic programming* (ILP) [Muggleton and de Raedt 1994], the goal is to learn a logic program from data, typically positive and negative examples of a target relation. ILP systems can learn from a small number of examples and with background knowledge (e.g., a set of horn clauses), and some systems are able to invent new predicates and learn programs with recursion [Cropper et al. 2020]. Typically, ILP systems learn Prolog programs, but recent work has explored learning in restricted hypothesis spaces for logic programs, e.g., Datalog [Albarghouthi et al. 2017; Evans and Grefenstette 2018] and answer set programming [Law et al. 2014]. As discussed in §9.1, our approach can be used to model some forms of ILP by encoding background knowledge in the grammar, and it seems possible that aspects of *metarules* [Muggleton et al. 2014] can also be achieved with our technique; exploring connections to ILP is an interesting avenue for future work.

11 CONCLUSION

We have argued for a very general tree automata-theoretic approach to learning logical formulas and, more generally, any expression which can be evaluated using state dependent on a background structure but independent of the expression size. This is the case for the finite variable logics studied in this work, as well as higher-order logics and logics with fixed point operators over finite structures. Precisely characterizing the power of this approach is an interesting direction for future work, and so too are the lower bounds and parameterized complexity questions we leave open.

What is nice about the tree automaton-based approach advocated here is that various infinite concept spaces constrained by a grammar can be seen to have finitely-many equivalence classes *modulo example structures.* Indeed, the states of the (minimal) automaton correspond to equivalence classes of formulas from the grammar that are equivalent with respect to the given input structures. Effective emptiness checking algorithms for tree automata show that we only need to keep a single representative from each equivalence class to solve synthesis. Exploring practical algorithms for restricted grammars and classes of structures, including learning in the presence of background theories (such as arithmetic, used say for counting), are intriguing directions for future work.

ACKNOWLEDGMENTS

We thank Victor Vianu for discussions and for suggesting the connection to logical types. This work was supported in part by a Discovery Partner's Institute (DPI) science team seed grant and a research grant from Amazon.

REFERENCES

Aws Albarghouthi, Paraschos Koutris, Mayur Naik, and Calvin Smith. 2017. Constraint-Based Synthesis of Datalog Programs. In *Principles and Practice of Constraint Programming*, J. Christopher Beck (Ed.). Springer International Publishing, Cham, 689–706. https://doi.org/10.1007/978-3-319-66158-2_44

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*. NATO Science for Peace and Security Series, D: Information and Communication Security, Vol. 40. IOS Press, 1–25.

Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. 2012. Synthesis of Reactive(1) designs. *J. Comput. System Sci.* 78, 3 (2012), 911–938. https://doi.org/10.1016/j.jcss.2011.08.007

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

Benjamin Caulfield, Markus N. Rabe, Sanjit A. Seshia, and Stavros Tripakis. 2015. What's Decidable about Syntax-Guided Synthesis? *CoRR* abs/1510.08393 (2015). http://arxiv.org/abs/1510.08393

Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing Database-Backed Applications with Query Synthesis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/2491956.2462180

Alonzo Church. 1960. Application of recursive arithmetic to the problem of circuit synthesis. *Summaries of talks presented at the Summer Institute for Symbolic Logic Cornell University, 1957, 2nd edn., Journal of Symbolic Logic* 28, 4 (1960), 30–50. 3a–45a.

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.

Andrew Cropper, Sebastijan Dumancic, and Stephen H. Muggleton. 2020. Turning 30: New Ideas in Inductive Logic Programming. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, Christian Bessiere (Ed.). ijcai.org, 4833–4839. https://doi.org/10.24963/ijcai.2020/673

Ugo Dal Lago. 2012. *A Short Introduction to Implicit Computational Complexity*. Springer Berlin Heidelberg, Berlin, Heidelberg, 89–109. https://doi.org/10.1007/978-3-642-31485-8_3

Andrzej Ehrenfeucht. 1961. An application of games to the completeness problem for formalized theories. *Fundamenta Mathematicae* 49, 2 (1961), 129–141. http://eudml.org/doc/213582

Herbert B. Enderton. 2001. Chapter 2 - First-Order Logic. In *A Mathematical Introduction to Logic* (second edition ed.), Herbert B. Enderton (Ed.). Academic Press, Boston, 67–181. https://doi.org/10.1016/B978-0-08-049646-7.50008-4

Richard Evans and Edward Grefenstette. 2018. Learning Explanatory Rules from Noisy Data. *J. Artif. Int. Res.* 61, 1 (Jan. 2018), 1–64. https://dl.acm.org/doi/10.5555/3241691.3241692

John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-Output Examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 229–239. https://doi.org/10.1145/2737924.2737977

J. Flum and M. Grohe. 2006. *Parameterized Complexity Theory (Texts in Theoretical Computer Science. An EATCS Series)*. Springer-Verlag, Berlin, Heidelberg. https://doi.org/10.1007/3-540-29953-X

Roland Fraïssé. 1953. *Sur quelques classifications des systèmes de relations*. Ph.D. Dissertation. Publications Scientifiques de l'Université d'Alger, series A 1 (1954).

Carsten Fritz. 2002. *Some Fixed Point Basics*. Springer Berlin Heidelberg, Berlin, Heidelberg, 359–364. https://doi.org/10.1007/3-540-36387-4_20

Maurice Funk, Jean Christoph Jung, Carsten Lutz, Hadrien Pulcini, and Frank Wolter. 2019. Learning Description Logic Concepts: When can Positive and Negative Examples be Separated?. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*. International Joint Conferences on Artificial Intelligence Organization, 1682–1688. https://doi.org/10.24963/ijcai.2019/233

Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. 2015. Quantified data automata for linear data structures: a register automaton model with applications to learning invariants of programs manipulating arrays and lists. *Formal Methods in System Design* 47, 1 (01 Aug 2015), 120–157. https://doi.org/10.1007/s10703-015-0231-6

Erich Grädel, Wolfgang Thomas, and Thomas Wilke (Eds.). 2002. *Automata Logics, and Infinite Games: A Guide to Current Research*. Springer-Verlag, Berlin, Heidelberg. https://doi.org/10.1007/3-540-36387-4

Martin Grohe, Christof Löding, and Martin Ritzert. 2017. Learning MSO-definable hypotheses on strings. In *Proceedings of the 28th International Conference on Algorithmic Learning Theory (Proceedings of Machine Learning Research, Vol. 76)*, Steve Hanneke and Lev Reyzin (Eds.). PMLR, Kyoto University, Kyoto, Japan, 434–451. http://proceedings.mlr.press/v76/grohe17a.html

Martin Grohe and Gy. Turán. 2004. Learnability and Definability in Trees and Similar Structures. *Theory of Computing Systems* 37, 1 (01 Jan 2004), 193–220. https://doi.org/10.1007/s00224-003-1112-8

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* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 317–330. https://doi.org/10.1145/1926385.1926423

Qinheping Hu, Jason Breck, John Cyphert, Loris D'Antoni, and Thomas Reps. 2019. Proving Unrealizability for Syntax-Guided Synthesis. In *Computer Aided Verification (Lecture Notes in Computer Science)*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 335–352. https://doi.org/10.1007/978-3-030-25540-4_18

Qinheping Hu, John Cyphert, Loris D'Antoni, and Thomas Reps. 2020. Exact and Approximate Methods for Proving Unrealizability of Syntax-Guided Synthesis Problems. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1128–1142. https://doi.org/10.1145/3385412.3385979

Neil Immerman. 1982. Relational Queries Computable in Polynomial Time (Extended Abstract). In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing* (San Francisco, California, USA) *(STOC '82)*. Association for Computing Machinery, New York, NY, USA, 147–152. https://doi.org/10.1145/800070.802187

Jean Christoph Jung, Carsten Lutz, Hadrien Pulcini, and Frank Wolter. 2020. Logical Separability of Incomplete Data under Ontologies. *CoRR* abs/2007.01610 (2020). https://arxiv.org/abs/2007.01610

Ayrat Khalimov, Benedikt Maderbacher, and Roderick Bloem. 2018. Bounded Synthesis of Register Transducers. In *Automated Technology for Verification and Analysis*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer International Publishing, Cham, 494–510. https://doi.org/10.1007/978-3-030-01090-4_29

Jason R. Koenig, Oded Padon, Neil Immerman, and Alex Aiken. 2020. First-Order Quantified Separators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 703–717. https://doi.org/10.1145/3385412.3386018

Paul Krogmeier and P. Madhusudan. 2021. Learning Formulas in Finite Variable Logics. arXiv:2111.03534 [cs.LO]

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. https://doi.org/10.1007/978-3-030-53291-8_32

Orna Kupferman, P. Madhusudan, P. S. Thiagarajan, and Moshe Y. Vardi. 2000. Open Systems in Reactive Environments: Control and Synthesis. In *CONCUR (Lecture Notes in Computer Science, Vol. 1877)*. Springer, 92–107. https://doi.org/10.1007/3-540-44618-4_9

Orna Kupferman, Nir Piterman, and Moshe Y. Vardi. 2010. *An Automata-Theoretic Approach to Infinite-State Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 202–259. https://doi.org/10.1007/978-3-642-13754-9_11

Mark Law, Alessandra Russo, and Krysia Broda. 2014. Inductive Learning of Answer Set Programs. In *Logics in Artificial Intelligence*, Eduardo Fermé and João Leite (Eds.). Springer International Publishing, Cham, 311–325. https://doi.org/10.1007/978-3-319-11558-0_22

Leonid Libkin. 2004. *Elements Of Finite Model Theory (Texts in Theoretical Computer Science. An Eatcs Series)*. SpringerVerlag. https://doi.org/10.1007/978-3-662-07003-1

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), Vol. 12)*, Marc Bezem (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 428–442. https://doi.org/10.4230/LIPIcs.CSL.2011.428

P. Madhusudan and P. S. Thiagarajan. 2001. Distributed Controller Synthesis for Local Specifications. In *ICALP (Lecture Notes in Computer Science, Vol. 2076)*. Springer, 396–407. https://doi.org/10.1007/3-540-48224-5_33

Zohar Manna and Richard Waldinger. 1980. A Deductive Approach to Program Synthesis. *ACM Trans. Program. Lang. Syst.* 2, 1 (Jan. 1980), 90–121. https://doi.org/10.1145/357084.357090

Umang Mathur, P. Madhusudan, and Mahesh Viswanathan. 2019. Decidable Verification of Uninterpreted Programs. *Proc. ACM Program. Lang.* 3, POPL, Article 46 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290359

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

Thomas M. Mitchell. 1997. *Machine Learning* (1 ed.). McGraw-Hill, Inc., USA.

Stephen Muggleton and Luc de Raedt. 1994. Inductive Logic Programming: Theory and methods. *The Journal of Logic Programming* 19-20 (1994), 629–679. https://doi.org/10.1016/0743-1066(94)90035-3

Stephen H. Muggleton, Dianhuan Lin, Niels Pahlavi, and Alireza Tamaddoni-Nezhad. 2014. Meta-interpretive learning: application to grammatical inference. *Machine Learning* 94, 1 (01 Jan 2014), 25–49. https://doi.org/10.1007/s10994-013-5358-3

Daniel Neider and Ivan Gavran. 2018. Learning Linear Temporal Properties. In *2018 Formal Methods in Computer Aided Design (FMCAD)*. 1–10. https://doi.org/10.23919/FMCAD.2018.8603016

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

Amir Pnueli and Roni Rosner. 1989. On the Synthesis of a Reactive Module. In *POPL*. ACM Press, 179–190. https://doi.org/10.1145/75277.75293

Amir Pnueli and Roni Rosner. 1990. Distributed Reactive Systems Are Hard to Synthesize. In *FOCS*. IEEE Computer Society, 746–757. https://doi.org/10.1109/FSCS.1990.89597

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 522–538. https://doi.org/10.1145/2908080.2908093

Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A Framework for Inductive Program Synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 107–126. https://doi.org/10.1145/2814270.2814310

Michael Oser Rabin. 1972. *Automata on Infinite Objects and Church's Problem.* American Mathematical Society, Boston, MA, USA. https://dl.acm.org/doi/10.5555/540412

Yanyan Shen, Kaushik Chakrabarti, Surajit Chaudhuri, Bolin Ding, and Lev Novik. 2014. Discovering Queries Based on Example Tuples. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 493–504. https://doi.org/10.1145/2588555.2593664

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS XII)*. Association for Computing Machinery, New York, NY, USA, 404–415. https://doi.org/10.1145/1168857.1168907

Aalok Thakkar, Aaditya Naik, Nathaniel Sands, Rajeev Alur, Mayur Naik, and Mukund Raghothaman. 2021. Example-Guided Synthesis of Relational Queries. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1110–1125. https://doi.org/10.1145/3453483.3454098

Steffen van Bergerem. 2019. Learning Concepts Definable in First-Order Logic with Counting. In *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 1–13. https://doi.org/10.1109/LICS.2019.8785811

Steffen van Bergerem, Martin Grohe, and Martin Ritzert. 2021. On the Parameterized Complexity of Learning Logic. *CoRR* abs/2102.12201 (2021). arXiv:2102.12201

Moshe Y. Vardi. 1982. The Complexity of Relational Query Languages (Extended Abstract). In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing* (San Francisco, California, USA) *(STOC '82)*. Association for Computing Machinery, New York, NY, USA, 137–146. https://doi.org/10.1145/800070.802186

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. https://doi.org/10.1007/BFb0055090

Victor Vianu. 2020. Personal communication.

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* (Barcelona, Spain) *(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

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017c. Synthesis of Data Completion Scripts Using Finite Tree Automata. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 62 (Oct. 2017), 26 pages. https://doi.org/10.1145/3133886

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