Languages With Decidable Learning: A Meta-Theorem

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

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

We study expression learning problems with syntactic restrictions and introduce the class of *finite-aspect checkable languages* to characterize symbolic languages that admit decidable learning. The semantics of such languages can be defined using a bounded amount of auxiliary information, which we call *semantic aspects*, that is independent of expression size but depends on a fixed structure over which evaluation occurs. We introduce a generic programming language for expressing model-checking programs that work over expression syntax trees, and we give a meta-theorem that connects such programs for finite-aspect checkable languages to finite tree automata, which allows us to derive new decidable learning results and decision procedures for several expression learning problems by writing model-checking programs in the programming language.

1 INTRODUCTION

We undertake a foundational theoretical exploration of the exact learning problem for *symbolic languages* with rich semantics. Learning symbolic concepts from data has myriad applications, e.g., in verification [Garg et al. 2014, 2015; Ivanov et al. 2021; Neider et al. 2020; Zhu et al. 2018] and, in particular, invariant synthesis for distributed protocols [Hance et al. 2021; Koenig et al. 2020, 2022; Yao et al. 2021], learning properties of programs [Astorga et al. 2019, 2021; Miltner et al. 2020], explaining executions of distributed protocols [Neider and Gavran 2018], and synthesizing programs from examples or specifications [Alur et al. 2015; Evans and Grefenstette 2018; Gulwani 2011; Handa and Rinard 2020; Muggleton et al. 2014; Polozov and Gulwani 2015; Solar-Lezama et al. 2006; Wang et al. 2017a,b].

In this paper, *symbolic languages* are construed as sets of expressions together with formal syntax and semantics. Languages include logics, e.g., first-order and modal logics, programming languages (functional or imperative), query languages like SQL, or even languages whose expressions describe other kinds of languages, e.g., regular expressions or context-free grammars. In the *exact learning problem* for a language $\mathcal{L}$, the goal is to find an expression $e \in \mathcal{L}$ that is consistent with a given finite set of (positively and negatively) labeled examples, which in this setting are *finite structures*. The expression e should be satisfied by all positive structures and not satisfied by any negative ones. The semantic notion, i.e. *satisfaction*, varies by problem.

Decidable Learning. The languages we study are complex enough that polynomial time learning is seldom possible (even learning the simplest Boolean formula that separates a labeled set of Boolean assignments to variables is not possible in polynomial time [Kearns and Vazirani 1994]). Furthermore, assuming the semantics of expressions over structures is computable (true in all languages we consider), there is always a trivial algorithm that *enumerates* expressions, evaluates them over the given structures, and finds a consistent expression if one exists. Given that enumeration can take exponential time in the size of the smallest consistent expression to terminate (if it terminates at all), and given any learning algorithm may require exponential time, a meaningful theoretical analysis of learning for such languages is hard. We hence consider *decidable learning*, where hypothesis classes are infinite and learning algorithms must terminate with a consistent expression if one

exists or report there is none. Note the trivial enumerator is not a decision procedure when there are infinitely-many expressions, since it may not be able to report an instance has no solution.

Learning under Syntactic Restrictions. We require learning algorithms to both *accommodate syntactic restrictions* over the language, i.e., restrictions to the hypothesis space, and to be able to find *small* expressions. These stipulations mitigate overfitting. For instance, in the case of some logics, the set of positively-labeled structures can be precisely captured using a single formula that can be computed efficiently given the structures. Such a solution is not interesting and is unlikely to generalize. In such cases, learning also becomes trivially decidable: if there is any consistent expression, then the highly specific one will be consistent. Accommodating syntactic restrictions and requiring small solutions circumvent these issues. Note also that syntactic restrictions are a *feature*— one can always allow expressions to be learned from the entire language.

Learning in Finite-Variable Logics. Our work draws inspiration from a recent result that showed classical logics, e.g., first-order logic (FO), have decidable learning when restricted to use finitely-many variables [Krogmeier and Madhusudan 2022]. The technique underlying this result uses *tree automata*. For each positively-labeled (respectively negatively-labeled) structure, one builds a tree automaton that reads expression syntax trees and accepts those that are true (respectively false) in that structure. Such automata are akin to *version space algebras* [Mitchell 1997], and taking their product (and a product with an automaton capturing the syntactic restriction) results in an automaton that accepts *all* consistent expressions. Existence of solutions can be decided with automata emptiness algorithms, which can be used to synthesize small consistent expressions if they exist.

Contributions. We show that the tree automata-theoretic technique for learning extends much beyond finite-variable logics. We prove decidable learning for a number of languages that are not finite-variable logics, and we give a meta-theorem that streamlines the task of proving decidable learning for new languages. It reduces proofs to the problem of writing an interpreter for the semantics in a particular *programming language*, which we call Facet. Using this meta-theorem, we exhibit a rich set of examples that have decidable learning:

- *Modal logic* over Kripke structures (Section 2)
- *Computation tree logic* over Kripke structures (Appendix C)
- *Regular expressions* over finite words (Section 5)
- *Linear temporal logic* over periodic words (Section 6)
- *Context-free grammars* over finite words (Section 7)
- *First-order queries* over tuples of rationals numbers with order (Section 8)
- *String transformations* from input-output examples (Section 9), similar to [Gulwani 2011]

In each of these settings, the learning problem is decidable under syntactic restrictions expressed by a (tree) grammar, which is given as an input along with the sample structures.

We emphasize our contributions are theoretical. The programming language itself is a notational tool that identifies and abstracts a pattern we observe many times in this work, that of *programming with two-way tree automata* to prove decidable learning and derive decision procedures. The learning algorithms we obtain have high complexity; implementing a compiler from the programming language to efficient decision procedures for learning will involve heuristics that depend on specific problems. We note that learning problems for several of the languages we study are of practical interest, with previous work exploring algorithms for regular expressions [Fernau 2009; Li et al. 2008], linear temporal logic [Neider and Gavran 2018], context-free grammars [Langley and Stromsten 2000; Sakakibara 2005; Vanlehn and Ball 1987], and string transformations in Microsoft Excel's Flash Fill [Gulwani 2011].

Meta-theorem. Each of the results above follows from a *meta-theorem* which says, intuitively, that languages are decidably learnable as long as expressions can be evaluated over any structure using a particular kind of program. More precisely, we require a *semantic evaluator P* that, given a structure M and expression e, evaluates e over M by navigating up and down on the syntax tree for e using recursion. Furthermore, P must rely only on a finite set of *semantic aspects of the structure* for memory during its navigation over e. This set depends on M but not on e. As long as we can write such an evaluator, the meta-theorem guarantees decidable learning for the language!

The notion of *semantic aspects* is quite natural in the settings we consider. In logic, the semantics of formulas φ over a structure M is often presented by recursion on φ, with some additional information. For instance, the satisfaction relation $M, \gamma \models \varphi$ for FO uses an interpretation of variables γ. For FO with a fixed set of k variables, the number of such γ is *bounded*; it depends on the structure M but not on the formula φ, and hence meets the finite-aspect requirement we identify in this paper. Consequently, the result on decidable learning for finite-variable FO is an immediate corollary of our meta-theorem. In fact, all such results for finite-variable logics [Krogmeier and Madhusudan 2022] are obtained as corollaries[1].

Semantic aspects are sometimes obvious from standard semantic definitions and sometimes less so. In *modal logic*, the standard semantics is defined recursively in terms of the semantics for subformulas at different *nodes* in a Kripke structure, and indeed, the aspects in this case are simply the nodes. For *computation tree logic* (CTL), standard semantics in the literature would use the nodes as for modal logic but would go beyond recursion in the structure of expressions and use recursive definitions to give meaning to formulas (least and greatest fixpoints [McMillan 1992]). In this case, the aspects include the nodes of the structure as well as a *counter* that encodes a recursion budget for *until* and *globally* formulas to be satisfied, with the counter value bounded by the number of nodes. The standard semantics for a *regular expression e* defines the language $L(e)$ recursively in the structure of e. For a given word w, however, we specialize the semantics of membership in $L(e)$ to w, using aspects that correspond to subwords of w (e.g., a pair of indices (i, j) marking the left and right endpoints of a subword, with $1 \leq i \leq j \leq |w|$). This membership semantics involves a finite number of aspects which is quadratic in the size of w but independent of e. Semantics of *linear temporal logic* (LTL) formulas over periodic words uv^ω can also be defined (non-standardly) using a set of aspects corresponding to each position of u and v, again finite. The semantics for membership of a word w in the yield of a *context-free grammar* (restricted to a finite set of nonterminals) can again be written with aspects corresponding to subwords, as for regular expressions. But in this case it also requires navigating the tree representing the grammar *up and down* many times in order to parse w, which requires keeping some extra memory. Standard semantics for *first-order queries* over rational numbers with order would involve an interpretation of variables as rational numbers, but this set is of course infinite. It turns out that a finite set of aspects encoding the *ordering* of the variables is sufficient to define semantics in this setting.

The meta-theorem is a powerful tool for establishing decidable learning. We emphasize that its proof is technically quite simple— programs that navigate trees using recursion can be translated to *two-way alternating tree automata*, which can be converted to one-way tree automata to obtain decision procedures for learning. Our technical contribution lies more in the formalization of the technique in terms of a programming language for semantic evaluators, and realizations in different settings with (nonstandard) semantic definitions involving a finite set of aspects.

We use the meta-theorem for the well-known application of learning string transfomations from examples in the context of spreadsheet programs. The seminal work of Gulwani [Gulwani

---

[1]The tree automata underlying each result for finite-variable logics can be easily translated to semantic evaluators of the kind we require. See Appendix D for such a semantic evaluator in the case of FO.

2011] established this problem as one of the first important applications of program synthesis from examples. We consider the language for string programs used in that work and argue that even a signficant extension of that language admits decidable learning. As far as we know, decidable learning for this well-studied problem was not known earlier.

Organization. In Section 2 we explore learning in modal logic to motivate the generic learning algorithm based on tree automata and the notion of semantic aspects. We discuss a semantic evaluator for modal formulas and abstract the main pattern as a program. Section 3 gives some background on tree automata. In Section 4 we define the class of finite-aspect checkable languages (languages that admit decidable learning), formalize a programming language for writing semantic evaluators, and give the meta-theorem connecting semantic evaluators to tree automata. Sections 5 to 8 establish decidable learning for regular expressions, linear temporal logic, context-free grammars, and first-order queries over rationals with order. In Section 9 we discuss decidable learning for string transformations. We review related work in Section 10 and conclude in Section 11.

2   MOTIVATING PROBLEM: LEARNING MODAL LOGIC FORMULAS

In this section, we illustrate how to derive learning algorithms from semantic evaluators in the context of propositional modal logic. We make the observation that a specific kind of semantic evaluator corresponds to a constructive proof of decidable learning. Such evaluators use an amount of memory *bounded by the structure* over which expressions are evaluated but independent of expression size, beyond that afforded by the syntax tree itself. To prove decidable learning for a new language, it suffices to program such an evaluator. We summarize this main theme as follows:

Effective evaluation using state bounded by structures $\implies$ *decidable learning*

We next introduce the learning problem for modal logic and explore this theme by developing a suitable semantic evaluator for modal formulas over Kripke structures.

2.1   Separating Kripke Structures with Modal Logic Formulas

Consider the following problem, with an example illustrated in Figure 1.

Problem (Modal Logic Separation). Given finite sets P and N of finite pointed Kripke structures over propositions Σ, and a grammar $\mathcal{G}$, synthesize a modal logic formula $\varphi \in L(\mathcal{G})$ that is true for structures in P (the *positives*) and false for those in N (*negatives*), or declare none exist.

We review some basics of modal logic [Blackburn et al. 2001]. The following grammar defines the set of modal logic formulas over a finite set of propositions Σ.

$$\varphi ::= a \in \Sigma \mid \varphi \wedge \varphi' \mid \varphi \vee \varphi' \mid \neg \varphi \mid \Box \varphi \mid \Diamond \varphi$$
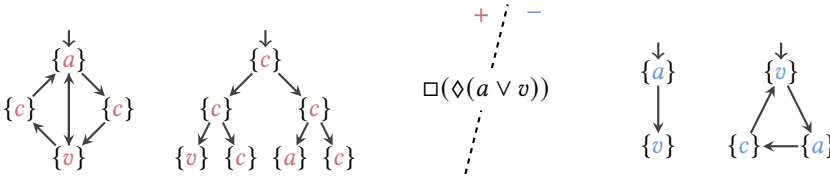

Fig. 1. The modal logic formula $\varphi = \Box(\Diamond(a \vee v))$ over $\Sigma = \{a, c, v\}$ is true for the two Kripke structures on the left and false for the two on the right. Starting nodes s are on top with incoming arrows.

The standard semantics of modal logic is reproduced below. Formulas are interpreted against (in our case finite) pointed Kripke structures $G = (W, s, E, P)$, where W is a set of nodes (or *worlds*),

E is a binary *neighbor* relation on W, and $P : W \rightarrow \mathcal{P}(\Sigma)$ is a function that labels each node by the set of all atomic propositions that hold there. A formula φ is true in $G = (W, s, E, P)$, written $G \models \varphi$, if it is true starting from s, written $G, s \models \varphi$, with the latter notion defined as follows.

$$
\begin{array}{llll}
G, w & \models & a \in \Sigma & \text{if} \quad a \in P(w) \\
G, w & \models & \neg\varphi & \text{if} \quad G, w \not\models \varphi \\
G, w & \models & \varphi \wedge \varphi' & \text{if} \quad G, w \models \varphi \text{ and } G, w \models \varphi' \\
G, w & \models & \varphi \vee \varphi' & \text{if} \quad G, w \models \varphi \text{ or } G, w \models \varphi' \\
G, w & \models & \Box\varphi & \text{if} \quad G, w' \models \varphi \text{ for all } w' \text{ such that } E(w, w') \\
G, w & \models & \Diamond\varphi & \text{if} \quad G, w' \models \varphi \text{ for some } w' \text{ such that } E(w, w')
\end{array}
$$

Observe that there are *infinitely-many* inequivalent modal formulas. Indeed, the sequence

$$\Diamond a, \ \Diamond(\Diamond a), \ \Diamond(\Diamond(\Diamond a)), \ \ldots$$

defines an infinite set $(\varphi_i)_{i \in \mathbb{N}^+}$ of semantically inequivalent formulas. For $i \in \mathbb{N}^+$, a finite graph consisting of a single directed path of length $i - 1$ makes the formula φ_i false while making all φ_j true for $j < i$. Thus the search space of modal formulas is infinite, and so we cannot resort to enumeration for decidable learning.

We advocate an automata-theoretic technique for learning problems of this kind, inspired by recent work on learning formulas in finite-variable logics [Krogmeier and Madhusudan 2022], which is summarized as follows:

(1) Encode language expressions as syntax trees over a finite alphabet.
(2) For each structure $p \in P$ (respectively, $n \in N$), construct a tree automaton accepting the syntax trees for expressions e such that $p \models e$ (respectively, $n \not\models e$).
(3) Construct a tree automaton accepting the intersection of languages for previous automata, which accepts all expressions consistent with the examples.
(4) Run an emptiness checking algorithm for the final automaton. If empty, report *unrealizable*. Otherwise, synthesize a (small) expression in the language of the automaton.

The procedure above adapts easily to learning with grammar restrictions. Given a regular tree grammar $\mathcal{G}$, we can construct a tree automaton accepting precisely the expressions allowed by $\mathcal{G}$ and take its intersection with the automaton from (3) before checking emptiness.

The crucial observation we make is that in order to apply this generic procedure to learning problems for new languages, we need only implement an *evaluator* for the semantics of the language. For any *fixed structure* M, the evaluator checks whether $M \models e$ for an input expression e, where "$\models$" is a problem-specific semantic relationship. Using the evaluator, we can compute the tree automaton for each given positive and negative structure and proceed with the algorithm above.

We can view these semantic evaluators as *programs* whose state depends on the mathematical structure over which evaluation occurs but depends only to a very small degree on the size of the expression itself. The key to finding these programs is to consider the question of how to interpret arbitrary input expressions from the language (presented as syntax trees) against an arbitrary, but *fixed*, structure. We invite the reader in the remainder of the section to naïvely explore how to write a program that evaluates an input modal logic formula φ against a fixed Kripke structure by traversing the syntax tree of φ.

2.2 Evaluating Modal Formulas on Fixed Kripke Structures

We want a procedure for evaluating any formula φ of modal logic against a fixed Kripke structure $G = (W, s, E, P)$, where *evaluate* means *verify* that $G \models \varphi$. The evaluator hence is designed for any particular G and takes the syntax tree of φ as input.

Imagine we want to evaluate the formula $\varphi = \Box(\Diamond(a \lor v))$ from Figure 1 over the rightmost positive structure G (tree-shaped). In particular, we want to check whether $G, s \models \varphi$ holds by traversing the syntax tree for φ (displayed on the right below) from the top down. Suppose n is a pointer into the syntax tree of φ, with n initially pointing to the root. We first read the symbol '$\Box$', and we recognize that $G, s \models \varphi$ holds exactly when the subformula $\Diamond(a \lor v)$ holds at each of the two children of s in G. Let w_1 and w_2 stand for the children of s, and let $c_i(n)$ stand for the ith child of the syntax tree pointed to by n. We now should recursively check whether $G, w_1 \models \Diamond(a \lor v)$ and $G, w_2 \models \Diamond(a \lor v)$ hold. To do this, we move *down* in the syntax tree by setting $n := c_1(n) = \Diamond(a \lor v)$. We then need to check $G, w_1' \models a \lor v$ holds, where w_1' is either the left or right child of w_1 in G (and likewise for w_2). Suppose we *nondeterministically guess* that $G, w_1' \models a \lor v$ holds with w_1' being the left child of w_1. We move *down* once more by setting $n := c_1(n) = a \lor v$ and we *verify* the guess by checking $G, w_1' \models a \lor v$. This plays out in a similar, straightforward way. Our traversal eventually terminates and returns true because $G, w_1' \models v$ holds, since $v \in P(w_1')$.
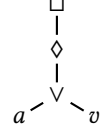
Note that the traversal described above works for arbitrary φ of unbounded size; indeed, the next steps are determined by the current symbol of the syntax tree pointed to by n and by some state that depends entirely on G, namely the set of nodes W, which is finite for a fixed, finite structure G. Observe also that the traversal required some computable functions specific to Kripke structures. For example, we needed to compute $P : V \to \mathcal{P}(\Sigma)$, membership for elements of $\mathcal{P}(\Sigma)$, and the set of neighbors of a given node in G under E.

2.3   A Program for Evaluating Modal Formulas

We conclude the modal logic example by writing a *program* which captures our traversal of φ and the computation of whether $G \models \varphi$. The program takes as inputs the structure G, some auxiliary state w, and a pointer n that initially points to the root of the syntax tree for a modal formula.

The program Modal is shown in Figure 2. We discuss formal semantics for such programs in Section 4; intuitively, the program implements the traversal sketched earlier by matching against the symbol $n.l$ that labels the current node of the syntax tree. Depending on the symbol, it can then either terminate by computing a Boolean function as its final answer (e.g. "$x \in P(w)$") or it can combine the results of recursive calls at nearby nodes on the syntax tree. It uses all and any to represent finite conjunctions and disjunctions, and it uses a few problem-specific computable functions, which we categorize as either *Boolean* functions or *state* functions. The only Boolean function in this case is for atomic propositions, i.e. "$z \in P(w)$", and the only state function is for computing the neighborhood of a given node in G, i.e. "$\{y \in G \ : \ E(w, y)\}$". Negation in Figure 2 is handled by evaluating the negated subformula in a dual state $dual(w)$, in which each part of the program is interpreted as its dual, e.g., and becomes or, *etc.* We return to this after formalizing the semantics for these programs.

Recall the theme from the beginning of this section:

Effective evaluation using state bounded by structures $\implies$ *decidable learning.*

The semantic evaluator for modal formulas uses auxiliary states that depend on the number of nodes in the Kripke structure, and *not* on the size of the syntax tree. Strictly speaking, it accesses the syntax tree using a pointer, and hence involves some minimal amount of memory that depends on expression size, but this is the *only* such dependence.

As we have just observed, effective evaluation of this sort is possible for modal logic on finite Kripke structures, and programs witnessing this fact like the one in Figure 2 imply decision procedures for learning. In the remainder of the paper, we define a class of languages that admit decidable learning and formulate the programming language that reduces decidable learning proofs

```
Modal(G, w, n) := match n.l with
    ∧  → Modal(G, w, n.c₁) and Modal(G, w, n.c₂)
    ∨  → Modal(G, w, n.c₁) or Modal(G, w, n.c₂)
    ¬  → Modal(G, dual(w), n.c₁)
    □  → all (λz. Modal(G, z, n.c₁)) {y ∈ G : E(w, y)}
    ◇  → any (λz. Modal(G, z, n.c₁)) {y ∈ G : E(w, y)}
    x  → x ∈ P(w)
```

Fig. 2. Modal evaluates modal formula φ pointed to by n against Kripke structure G and checks $G \models \varphi$.

to programming a semantic evaluator. We use the language to obtain results for several other learning problems.

3 PRELIMINARIES

Here we review some background on syntax trees, tree grammars, and tree automata.

3.1 Syntax Trees and Tree Grammars

For each symbolic language in this paper we use a *ranked alphabet* to form expression syntax trees. A ranked alphabet Δ is a set of symbols s equipped with a function $\mathrm{arity}(s) \in \mathbb{N}$. For example, the ranked alphabet for modal formulas over Σ has $\mathrm{arity}(\diamond) = 1$, $\mathrm{arity}(\wedge) = 2$, and $\mathrm{arity}(a) = 0$ for each $a \in \Sigma$. We write T_Δ for the set of Δ-terms, or $(\Delta\text{-})syntax\ trees$, which is the smallest set containing symbols of arity 0 from Δ and closed under forming new terms with symbols of greater arities. We write $T_\Delta(X)$ for the set of Δ-terms constructed with a fresh set of nullary symbols X.

We use regular tree grammars to express syntax restrictions for learning problems. A *regular tree grammar* is a tuple $\mathcal{G} = (NT, \Delta, S, P)$ consisting of a finite set of nonterminals NT, ranked alphabet Δ, starting nonterminal $S \in NT$, and productions P. Each production has the form "$A \to t$", with $A \in NT$ and $t \in T_\Delta(NT)$. In a standard way we associate with the productions P a reflexive and transitive rewrite relation $\to_P^*$ on terms $T_\Delta(NT)$, and the language $L(\mathcal{G})$ is the set $\left\{ t \in T_\Delta(\emptyset) \mid S \to_P^* t \right\}$. See [Comon et al. 2007] for details.

3.2 Tree Automata

Tree automata are finite state machines that accept trees. We use tree automata over *finite* trees, or terms. Such automata are tuples $\mathcal{A} = (Q, \Delta, q_i, \delta, F)$ consisting of a finite set of states Q, ranked alphabet Δ, initial state $q_i \in Q$, transition function δ, and acceptance condition F. An automaton accepts a tree $t \in T_\Delta$ if it has an *accepting run* over t. The notions of *run* and *accepting run* can vary.

In this work we use a convenient, though no more expressive, variant of tree automata called an *alternating two-way tree automaton*. Such automata walk up and down on their input tree and branch using alternation to send copies of the automaton in updated states to nearby nodes of the tree. We will only use *reachability* acceptance conditions in this paper, where $F \subseteq Q$, and a tree is accepted if along every trajectory of the automaton during its walk over the tree, it reaches a state in F. We omit the formal definition of *runs* for these automata, which is entirely standard, though complicated, and unnecessary for understanding our results.

For a symbol $s \in \Delta$ with $\mathrm{arity}(s) = k$ and state $q \in Q$, the available transitions for a two-way alternating tree automaton are described by a Boolean formula

$$\delta(q, s) \in \mathcal{B}^+(Q \times \{-1, 0, \ldots, k\}),$$

where $\mathcal{B}^+(X)$ means the set of positive Boolean formulas over variables from a set X. Each variable (q, m) represents a new state q and direction m to take at a particular node in the tree, with $m = -1$ being a move *up* to the parent of the current node, $m = 0$ meaning to *stay* at the current node, and the other numbers being moves down into one of k children. A subset of $Q \times \{-1, 0, \ldots, k\}$ corresponds to a Boolean assignment, and the automaton can proceed according to any assignment that satisfies the current transition formula. For example, if the automaton reads symbol h in state q, the transition

$$\delta(q, h) = (q_1, 1) \wedge (q_2, 1) \vee (q_1, 2) \wedge (q_2, 0) \wedge (q_1, -1)$$

would allow either of the following: (1) continuing in states q_1 and q_2, each starting from the leftmost child, or (2) continuing from q_1 in the second child from left, from q_2 at the current node, and from q_1 in the parent.

Two-way alternating tree automata can be converted to one-way nondeterministic tree automata with an exponential increase in states [Cachat 2002; Vardi 1998], and so they inherit closure properties and standard decision procedures. In particular, the emptiness problem can be solved in exponential time and a small tree in the language can be synthesized in the same amount of time when nonempty. See [Comon et al. 2007] for details.

4   DECIDABLE LEARNING VIA PROGRAMMING SEMANTIC EVALUATORS

In this section we define a rich class of languages for which decidable learning is possible. We then develop a meta-theorem which enables proofs of decidable learning by writing semantic evaluators in a programming language, which we call Facet[2]. The decision procedures involve an effective translation of Facet programs into two-way alternating tree automata that read syntax trees. After defining the class (Section 4.1), we discuss the syntax and semantics of Facet (Section 4.2), followed by the meta-theorem (Section 4.3), which says that all languages whose semantics can be written in Facet are decidably learnable. We then apply this theorem to show decidable learning for modal logic (Section 4.4) and computation tree logic (Appendix C).

4.1   A Class of Languages with Decidable Learning

There is a surprisingly rich set of languages that have decidable learning via essentially one generic decision procedure, which we describe here. For our purposes, a *language* consists of a set of expressions $\mathcal{L}$[3], a class of finitely-representable structures $\mathcal{M}$ over the same signature, and a semantic function that interprets a structure and an expression in some domain D, written with a turnstile as $(\_ \models \_) : \mathcal{M} \times \mathcal{L} \to D$. Sometimes we just use $\mathcal{L}$ to refer to such a symbolic language.

The decision procedure relies on building a tree automaton that accepts the set of all (syntax trees for) expressions $e \in \mathcal{L}$ that are consistent with a given example. In a supervised learning scenario, with $D = \mathbb{B} := \{\text{True}, \text{False}\}$ and examples modeled as pairs $(M, b) \in \mathcal{M} \times \mathbb{B}$, one builds a tree automaton $\mathcal{A}(M, b)$ such that $L(\mathcal{A}(M, b)) = \{e \in \mathcal{L} : M \models e = b\}$. For a finite set of examples $E = (M_i, b_i)_i$, we take the product $\mathcal{A}(E) = \bigwedge_i \mathcal{A}(M_i, b_i)$. Given an automaton $\mathcal{A}(\mathcal{G})$ that accepts syntax trees conforming to a tree grammar $\mathcal{G}$, we construct the product $\mathcal{A}(E) \wedge \mathcal{A}(\mathcal{G})$ and run emptiness algorithms to synthesize a syntax tree in the language or declare none exist.

The crucial requirement above is to be able to build $\mathcal{A}(M, b)$ for any M, which is an automaton that acts as an evaluator for the language over M. This is possible when the semantics of a language is definable in terms of a *finite* amount of auxiliary information, which may depend (sometimes wildly) on the particular structure M but not on the expression size. We refer to such auxiliary

---

[2]Facet stands for finite aspect checkers of expression trees.
[3]We often abuse notation and do not distinguish between expressions $e \in \mathcal{L}$ and the syntax trees for e.

semantic information as *semantic aspects*, or just *aspects*, and we call languages for which evaluators can be implemented using tree automata *finite-aspect checkable*[4].

Definition 4.1 (Finite-Aspect Checkable Language). A language $(\mathcal{M}, \mathcal{L}, \models)$ is *finite-aspect checkable* (FAC) if for every $(M, d) \in \mathcal{M} \times D$ there is a tree automaton $\mathcal{A}(M, d)$ over syntax trees for $\mathcal{L}$ such that $L(\mathcal{A}(M, d)) = \{e \in \mathcal{L} \: : \: M \models e = d\}$, and the mapping $(M, d) \mapsto \mathcal{A}(M, d)$ is computable.

Note that *all* FAC languages have decidable learning by the generic algorithm described above.

FAC languages only require the automata to be computable given $(M, d) \in \mathcal{M} \times D$, but all examples we have considered in fact have small *witnesses* for being FAC: the tree automata can be described compactly by a *program* that evaluates an input syntax tree against an input structure. We next describe the programming language FACET, which abstracts the common features of such programs. Our meta-theorem relies on a simple procedure that takes a program $P \in$ FACET, a structure $M \in \mathcal{M}$, and a domain element $d \in D$, and computes the tree automaton $\mathcal{A}(M, d)$.

4.2 Syntax and Semantics of FACET

We present the syntax and semantics of FACET by way of example. We omit many details that are not important for understanding later sections and results; details, including formal semantics of FACET, can be found in Appendix A.

Programs in FACET are parameterized by a symbolic language $(\mathcal{L}, \mathcal{M}, \models)$. A program P takes as input a *pointer* into the syntax tree for an expression $e \in \mathcal{L}$ as well as a structure $M \in \mathcal{M}$. The program navigates up and down on e using a set of pointers to move from children to parent and parent to children in order to evaluate the semantics of e over the structure M and verify that $M \models e = d$ for some $d \in D$. To write a FACET program we first specify two things: (1) the symbolic language $\mathcal{L}$ over which the program is to operate and (2) the program's auxiliary state, which corresponds to the semantic aspects of $\mathcal{L}$. Part (1) involves specifying (1a) the syntax trees for $\mathcal{L}$ in terms of a ranked alphabet Δ and (1b) the signature for structures $\mathcal{M}$, which is a set of functions used to access the data for any given $M \in \mathcal{M}$. The set of auxiliary states of part (2), which we denote by Asp, will typically be infinite. For a fixed structure $M \in \mathcal{M}$, however, programs will only use a finite subset $\text{Asp}(M) \subset \text{Asp}$, provided the symbolic language $\mathcal{L}$ is FAC, and so we will only need to specify $\text{Asp}(M)$ for an arbitrary fixed M.

For example, consider the program Modal for modal logic in Figure 2. Part (1): the symbolic language is modal logic over finite pointed Kripke structures $G = (W, s, E, P)$ with propositions Σ. We fix any straightforward representation of formulas as syntax trees, and we use two Kripke structure-specific functions for interpreting modal logic. The first computes the neighborhood $\{y \in G \: : \: E(w, y)\}$ of a given node $w \in W$, and the second computes whether a given proposition $x \in \Sigma$ is true at a given node $w \in W$, i.e. whether $x \in P(w)$ holds. Part (2): the states $\text{Asp}(M)$ are the nodes of the Kripke structure, i.e. the set W.

4.2.1 Syntax. The formal syntax for FACET programs is shown in Figure 3. A program $P \in$ FACET consists of a set of *clauses*, which we denote by $C(P)$, or just C, each of which has the form

$$P(M, \ \sigma(z), \ n) \ \coloneqq \ \texttt{match } n.l \texttt{ with } ...$$

The parameter M is a mathematical structure (e.g. a Kripke structure) and the parameter n is a pointer into a syntax tree (e.g. for a modal logic formula). The parameter $\sigma(z)$ is a *pattern* (e.g. a variable w matching any node of a Kripke structure). We treat both the ranked alphabet Δ and auxiliary states $\text{Asp}(M)$ as algebraic data types and allow FACET programs to pattern match over these using expressions from two sets of patterns, alphabet patterns $pat(\Delta)$ and state patterns

---

[4]*Checkable* refers to the *model checking* problem for a logic, i.e. checking whether $M \models \varphi$ for a structure M and formula φ.

$$
\begin{array}{lll}
\textit{Prog} & ::= & \{\, \textit{Clause} \ldots \textit{Clause} \,\} \\
\textit{Clause} & ::= & \texttt{P}(M,\ \sigma(z),\ n)\ \coloneqq\ \texttt{match}\ n.l\ \texttt{with}\ \textit{Cases} \\
\textit{Cases} & ::= & \alpha_1(z)\ \rightarrow\ e_1\ \ldots\ \alpha_n(z)\ \rightarrow\ e_n
\end{array}
$$

$$
\begin{array}{l}
e\ ::=\ \texttt{True} \qquad\qquad\quad |\ \ \texttt{False} \qquad\qquad\quad |\ \ f(z) \\
\quad |\ \ e_1\ \texttt{and}\ e_2 \qquad\quad |\ \ e_1\ \texttt{or}\ e_2 \qquad\quad\ \ |\ \ \texttt{P}(M,\ \sigma(z),\ n.dir) \\
\quad |\ \ \texttt{all}\ (\lambda x.\ e)\ g(z) \quad |\ \ \texttt{any}\ (\lambda x.\ e)\ g(z) \quad |\ \ \texttt{if}\ f(z)\ \texttt{then}\ e_1\ \texttt{else}\ e_2
\end{array}
$$

$$
\alpha(z) \in pat(\Delta) \qquad \sigma(z) \in pat(\text{Asp}) \qquad f \in B,\ g \in S \qquad dir \in \{up,\, stay,\, c_1,\, ...,\, c_k\}
$$

Fig. 3. Syntax for FACET programs. We use x to denote a single variable and z to denote a vector of variables.

$pat(\text{Asp})$. For example, for Modal in Figure 2, the (trivial) state pattern $w \in pat(\text{Asp})$ will match any node of the input Kripke structure, and the (trivial) alphabet pattern $x \in pat(\Delta)$ will match any of the modal logic propositions in Σ.

Each clause in C has a single match statement, consisting of a list of *cases*, each of the form "$\alpha_i(z) \rightarrow e$", with an alphabet pattern on the left and an expression on the right. Expressions e represent Boolean functions, possibly involving results of recursive calls "$\texttt{P}(M, \sigma(z), n.dir)$" that start in new states $\sigma(z)$ at nearby nodes $n.dir$ on the syntax tree. Compound expressions are built using and, or, all, any, and if. For example, in Figure 2, the first two cases involve recursive calls at the two children ($n.c_1$ and $n.c_2$) of the current node, in the same state w, and return, respectively, the conjunction and disjunction of the results.

The sets S and B in Figure 3 categorize the signature functions for structures $\mathcal{M}$ as one of two kinds. *State* functions $g \in S$ are used to compute new states for the program, e.g. computing the neighborhood of a given node in a Kripke structure. These functions are used in any and all expressions to bind parts of the structure M to variables. *Boolean* functions[5] $f \in B$ are used in if expressions as well as for base cases in recursion, e.g. computing membership in the set of true propositions for each node of a Kripke structure. Note that for all the symbolic languages studied in this paper, the signature functions are evidently computable.

4.2.2 Semantics. Given a structure M, state $\sigma \in \text{Asp}(M)$, and pointer n, a program operates by first determining the clause whose state pattern $\sigma(z)$ matches σ. We consider only *well-formed* programs in which every state is matched by the state pattern of precisely one clause (see Appendix A.0.3 for details). After the clause is determined, the program matches the symbol $n.l$ labeling the current node of the syntax tree against the alphabet patterns. It then evaluates the expression on the right side of the first matching case and returns a Boolean result, either success or failure.

Consider the operation of the program Modal from Figure 2 over the tree-shaped positive Kripke structure from Figure 1 and the formula $\varphi = \Box(\Diamond(a \lor v))$. Suppose n is pointing at the root of φ, with label $\Box$, and the current state is $s \in W$ (the top node of the Kripke structure). The state s matches the state pattern of the clause depicted in Figure 2, and the variable w is bound to s. Next, the symbol $n.l = \Box$ matches the fourth case of the match statement, and the program evaluates the all expression. To do this, it uses a state function to compute the neighborhood $N(s) = \{y \in G : E(s, y)\}$. It then returns the conjunction of results for recursive calls obtained by evaluating $\texttt{Modal}(G, z, n.c_1)$ with z bound to the states of $N(s)$. This involves two recursive calls with pointer $n.c_1$ corresponding to the formula $\Diamond(a \lor v)$ in states corresponding to the two neighbors of s. Each of these recursive calls involves evaluating the any expression for the $\Diamond$ case, which in

---

[5]We assume the set of Boolean functions B is closed under complement.

turn involves evaluating the expression for the $\vee$ case. Finally, when the program encounters either of the propositions $a, v \in \Sigma$ in the syntax tree, the last case will match, and the program uses a function to compute $x \in P(w)$, i.e. whether the proposition bound to x is true at the current state.

Formal semantics for FACET can be found in Appendix A. The semantics defines a predicate $\Downarrow_p$ on program configurations of the form $(M, n, C(P), \sigma)$. The assertion $(M, n, C(P), \sigma) \Downarrow_p$ has the following meaning: the program consisting of clauses $C(P)$ terminates with success on the syntax tree pointed to by n, when started from state σ over the structure M.

4.3 A Meta-Theorem for Decidable Learning

We now connect FACET programs with automata to give our meta-theorem for decidable learning. Its proof relies on the following lemma, which states that semantic evaluators written in FACET that use finite auxiliary states for any structure can be translated to two-way alternating tree automata.

LEMMA 4.2. *Let* $(\mathcal{L}, \mathcal{M}, \models)$ *be a symbolic language and* Δ *an alphabet for* $\mathcal{L}$. *Let P be a well-formed* FACET *program over* $(\mathcal{L}, \mathcal{M}, \models)$ *with computable signature functions and with* $\mathrm{Asp}(M)$ *finite for every $M \in \mathcal{M}$. Then for every $M \in \mathcal{M}$ and $\sigma \in \mathrm{Asp}(M)$, we can compute a two-way alternating tree automaton $\mathcal{A}(P, M, \sigma) = (Q, \Delta, q_i, \delta, F)$ such that for every $e \in \mathcal{L}$, we have $e \in L(\mathcal{A}(P, M, \sigma))$ if and only if $(M, root(e), C(P), \sigma) \Downarrow_p$.*

PROOF SKETCH. The automaton states are $Q = \mathrm{Asp}(M) \sqcup \{q_\top, q_\bot\}$, with $q_\top$ and $q_\bot$ being absorbing states for accepting and rejecting upon termination of the program. The initial state is $q_i = \sigma$, and the transitions δ are obtained from a straightforward translation of expressions into Boolean formulas, detailed in Appendix A.1. The acceptance condition is reachability with $F = \{q_\top\}$. The correspondence between the language of the automaton and semantic proofs for P is straightforward and essentially follows by construction. □

THEOREM 4.3 (META-THEOREM). *Let* $(\mathcal{L}, \mathcal{M}, \models)$ *be a language. If there is a semantic evaluator, i.e., a well-formed program $P \in$ FACET, such that for all $M \in \mathcal{M}$, $e \in \mathcal{L}$, and $d \in D$ there is a $\sigma_d \in \mathrm{Asp}(M)$ for which $(M, root(e), C(P), \sigma_d) \Downarrow_p$ if and only if $M \models e = d$, then $(\mathcal{L}, \mathcal{M}, \models)$ has decidable learning.*

PROOF. Let P be a semantic evaluator for a language $(\mathcal{L}, \mathcal{M}, \models)$, let $(M_i, d_i)_i$ be a finite set of examples from $\mathcal{M} \times D$, and let $\mathcal{G}$ be a (tree) grammar for $\mathcal{L}$. Use Lemma 4.2 to build the tree automata $\mathcal{A}(P, M_i, \sigma_{d_i})$. Construct the product of these automata and convert the result to a nondeterministic tree automaton $\mathcal{A}$. Take the product of $\mathcal{A}$ with a nondeterministic tree automaton for $\mathcal{G}$, and use an emptiness algorithm to synthesize an expression or decide there is none. □

Remark on Complexity. Notice that the states of the resulting automaton are precisely what we have called semantic aspects. The complexity of decision procedures for learning can in fact be read from the number of aspects. Products of two-way tree automata obtained from Lemma 4.2 are converted to one-way nondeterministic tree automata with an exponential increase in states using known algorithms [Cachat 2002; Vardi 1998], leading to decision procedures with time complexity exponential in the number of aspects as well as the number of examples. Provided that signature functions are computable in time exponential in the size of structures (satisfied by all languages in this paper), we can use the following:

COROLLARY 4.4. *Decision procedures for learning obtained via Theorem 4.3 have time complexity exponential in the number of examples and the number of aspects, and linear in the size of the grammar.*

FACET enables compact, familiar descriptions of two-way tree automata, and thereby enables decision procedures for learning to be developed using intuition from programming. If we can

```
Modal(G, dual(w), n) := match n.l with
      ∧  → Modal(G, dual(w), n.c₁) or Modal(G, dual(w), n.c₂)
      ∨  → Modal(G, dual(w), n.c₁) and Modal(G, dual(w), n.c₂)
      ¬  → Modal(G, w, n.c₁)
      □  → any (λz. Modal(G, dual(z), n.c₁)) {y ∈ G : E(w, y)}
      ◊  → all (λz. Modal(G, dual(z), n.c₁)) {y ∈ G : E(w, y)}
      x  → x ∉ P(w)
```

Fig. 4. Dual clause for Modal, which evaluates formula φ pointed to by n against G and verifies $G \not\models \varphi$.

write a FACET program to interpret a language $\mathcal{L}$ using a fixed amount of auxiliary state for any given structure, then the language is FAC and decision procedures for learning and synthesis follow from results in automata theory.

4.4 Decidable Learning for Modal Logic and Dual Clauses

We finish this section with a decidable learning theorem for modal logic by completing the Modal program from Section 2 and explaining *dual* states and programs, which are syntactic sugar useful for handling negation and negative examples.

The grammar for modal logic formulas over propositions Σ from earlier has a straightforward ranked alphabet Δ, with members of Σ having arity 0. The class $\mathcal{M}$ consists of finite pointed Kripke structures $G = (W, s, E, P)$. The states for a given $G = (W, s, E, P)$ are

$$\mathsf{Asp}(G) = \{w, dual(w) \; : \; w \in W\},$$

where *dual* is a constructor for states related to negation. There are two signature functions: a state function for computing neighborhoods $\{y \in G : E(w, y)\}$ and a Boolean function for computing membership of propositions in $P(w)$, for a given $w \in W$. Along with the clause in Figure 2, Modal includes the *dual* of that clause, shown in Figure 4, which operates on states of the form $dual(x)$. For any clause c there is a simple translation to produce its dual clause $dual(c)$ as follows:

```
        c  =  P(M, σ(z), n)       := match n.l with α₁ → e₁       ... αₙ → eₙ
  dual(c) =  P(M, dual(σ(z)), n)  := match n.l with α₁ → dual(e₁) ... αₙ → dual(eₙ)
```

The expressions $dual(e_i)$ are obtained by recursively swapping True with False, and with or, and all with any, and the dual and non-dual clauses invoke each other whenever a negation operator is encountered in the syntax tree. Dual clauses are useful even if the language $\mathcal{L}$ has no negation operator, given we may need to check that semantic relationships *do not* hold for negative structures. Note that the concept of dual clause is independent of the symbolic language $\mathcal{L}$, and it is entirely mechanical to obtain dual clauses in each problem we consider. A precise description of $dual(e)$ can be found in Appendix B. From now on we omit the dual clauses from our presentation.

THEOREM 4.5. *Modal logic separation for sets of Kripke structures P and N with grammar $\mathcal{G}$ is decidable in time $\mathcal{O}(2^{poly(mn)} \cdot |\mathcal{G}|)$, where $n = \max_{G \in P \cup N} |G|$ and $m = |P| + |N|$.*

PROOF SKETCH. For all Kripke structures $G = (W, s, E, P)$, $w \in W$, and formulas φ, we have that $G, w \models \varphi$ iff $(G, root(\varphi), C(\mathsf{Modal}), w) \Downarrow_p$ and $G, w \not\models \varphi$ iff $(G, root(\varphi), C(\mathsf{Modal}), dual(w)) \Downarrow_p$. The proof is by induction on φ. The rest follows by Theorem 4.3 and Corollary 4.4, with $D = \{\mathsf{True}, \mathsf{False}\}$, $\sigma_{\mathsf{True}} = s$ and $\sigma_{\mathsf{False}} = dual(s)$, noting that $|\mathsf{Asp}(G)| = \mathcal{O}(|W|)$.   □

Computation tree logic. Do other modal logics have decidable learning? For computation tree logic (CTL) the answer is affirmative, and we can program a FACET evaluator whose semantic aspects again involve the nodes of the Kripke structure. CTL involves, however, quantification over *paths* in the Kripke structure, with semantics given by recursive definitions like

$$\mathsf{E}(\varphi \mathsf{U} \varphi') \equiv \varphi' \vee (\varphi \wedge \mathsf{E}(\mathsf{X}(\mathsf{E}(\varphi \mathsf{U} \varphi'))))$$

for the existential path quantifier E. This introduces a subtlety related to negation, because the evaluator should avoid infinite recursion caused by interpreting E as the right-hand side of the equivalence above. We include in Appendix C a program for CTL that uses a *bounded counter* to prevent such infinite recursion.

In the remainder of the paper we derive new decision procedures for several learning problems by writing FACET programs. We avoid details about the FACET language and focus instead on the logic of the programs as well as the semantic aspects needed to accurately evaluate expressions.

5 LEARNING REGULAR EXPRESSIONS

In this section we develop a decision procedure for learning regular expressions from finite words. In contrast to propositional modal logic, the semantics of regular expressions involves recursion in the structure of expressions as well as recursion over the structures themselves.

5.1 Separating Words with Regular Expressions

Consider the following problem.

Problem (Regular Expression Separation). Given finite sets P and N of finite words over an alphabet Σ, and a grammar $\mathcal{G}$, synthesize a regular expression over Σ that matches all words in P, does not match any word in N, and conforms to $\mathcal{G}$, or declare none exist.

We consider (extended) regular expressions from the following grammar.

$$e ::= a \in \Sigma \mid e \cdot e' \mid e + e' \mid e \cap e' \mid e^* \mid \neg e$$

Recall that we want to program an evaluator for regular expressions over fixed Σ-words. The notion of evaluation here is *membership* of a word w in the language of a regular expression e, i.e. $w \models e \Leftrightarrow w \in L(e)$. This semantics has a straightforward recursive definition, and it can be presented in terms of an auxiliary relation that makes the finite semantic aspects plain to see:

$$w \in L(e) \quad \Leftrightarrow \quad w, (1, |w| + 1) \models e.$$

This definition uses the aspect of *subwords* of w, with (l, r) indicating the subword $w(l, r)$ that includes the letter at position l and extends to the position before r. Note that the empty word ϵ can therefore be represented by (i, i) for any i. The semantics of subword membership is given below.

$$
\begin{array}{llllll}
w, (l, r) & \models & a \in \Sigma & \text{if} & w(l) = a & \text{and} \quad r = l + 1 \\
w, (l, r) & \models & e \cdot e' & \text{if} & w, (l, k) \models e & \text{and} \quad w, (k, r) \models e' \quad \text{for some } k \in [l, r] \\
w, (l, r) & \models & e + e' & \text{if} & w, (l, r) \models e & \text{or} \quad w, (l, r) \models e' \\
w, (l, r) & \models & e \cap e' & \text{if} & w, (l, r) \models e & \text{and} \quad w, (l, r) \models e' \\
w, (l, r) & \models & e^* & \text{if} & l = r & \text{or} \quad \exists k \in [l + 1, r]. \ w, (l, k) \models e \ \text{and} \ w, (k, r) \models e^* \\
w, (l, r) & \models & \neg e & \text{if} & w, (l, r) \not\models e
\end{array}
$$

Observe that if we *fix* the word w then the number of pairs (l, r) used in the definition above is finite. Also observe that the definition in the case for Kleene star is well-founded because either the expression size decreases or the subword length decreases.

```
Reg(w, (l,r), n) := match n.l with
   *   → if (l = r) then True else
            any (λx. Reg(w, (l,x), n.c₁) and Reg(w, (x,r), n.stay)) [l + 1,r]
   ·   → any (λx. Reg(w, (l,x), n.c₁) and Reg(w, (x,r), n.c₂)) [l,r]
   +   → Reg(w, (l,r), n.c₁) or Reg(w, (l,r), n.c₂)
   ¬   → Reg(w, dual(l,r), n.c₁)
   ∩   → Reg(w, (l,r), n.c₁) and Reg(w, (l,r), n.c₂)
   x   → r = l + 1 and w(l) = x
```

Fig. 5. Reg evaluates the regular expression e pointed to by n against an input word w and verifies $w \in L(e)$.

Remark. Regular expression separation has two overfitting-style solutions if we ignore the syntax restriction from the input grammar $\mathcal{G}$. Simply use $+_{w \in P} w$ for the tightest regular expression that matches all of P, or alternatively, $\cap_{w \in N} \neg w$ for the loosest one that avoids matching any of N. If there is any separating regular expression at all, then either of these must work.

5.2 Decidable Learning for Regular Expressions

We write a Facet program called Reg which reads a regular expression syntax tree and verifies whether a given word is a member of the language for the regular expression. The class $\mathcal{M}$ consists of structures encoding Σ-words, $\mathcal{L}$ consists of regular expressions over Σ, and semantics is membership in the language of regular expressions. States are pairs of ordered indices, representing subwords, along with duals to handle negation.

$$\mathsf{Asp}(w) := \{(l,r), dual(l,r) \,:\, l \leq r \in [1, |w| + 1]\}.$$

For instance, if $w = abbb$, then the subword $w' = ab$ is represented as the pair of positions $(1, 3)$. The alphabet Δ for syntax trees is straightforward and uses symbols of arity 0 for members of Σ. We use state functions for looking up the letter at a given position i, written $w(i)$, the successor function on positions x, written $x + 1$, and functions $[x, y]$ and $[x + 1, y]$ for computing the indices between two positions $x \leq y$, with $[x + 1, y] = \emptyset$ if $x = y$. Boolean functions include equality and disequality on positions and letters of Σ.

The program Reg (with dual omitted) is given in Figure 5. States matching (l, r) are used by the program to check whether $w(l, r) \in L(e)$, and states matching $dual(l, r)$ are used to check whether $w(l, r) \notin L(e)$. Using Reg we get the following.

Theorem 5.1. *Regular expression separation for sets of words P and N and grammar $\mathcal{G}$ is decidable in time $\mathcal{O}(2^{poly(mn^2)} \cdot |G|)$, where $n = \max_{w \in P \cup N} |w|$ and $m = |P| + |N|$.*

Proof Sketch. For all words w, positions $1 \leq i \leq j \leq |w| + 1$, and regular expressions e, we have that $w(i, j) \in L(e)$ if and only if $(w, root(e), C(\text{Reg}), (i, j)) \Downarrow_p$ and $w(i, j) \notin L(e)$ if and only if $(w, root(e), C(\text{Reg}), dual(i, j)) \Downarrow_p$. The proof is by induction on $|w|$ and inner induction on e. We have $|\mathsf{Asp}(w)| = \mathcal{O}(|w|^2)$, and the theorem follows by Theorem 4.3 and Corollary 4.4. □

6 LINEAR TEMPORAL LOGIC

In this section we consider synthesizing linear temporal logic (LTL) formulas that separate infinite, periodic words. We again derive a decision procedure for learning by writing a program.

6.1 Separating Infinite Words with Linear Temporal Logic

We consider separating lassos over a finite alphabet Σ. A *lasso* is an infinite periodic word $w \in \Sigma^\omega$ that can be represented finitely as the concatenation of a finite prefix $u \in \Sigma^*$ with a finite *repeated* suffix $v \in \Sigma^*$. For example, the infinite word $babaabbaabbaabbaabb\cdots$ can be represented with $u = bab$ and $v = aabb$. Lassos are determined by the pair (u, v), though there may be multiple ways to pick u and v. Here we could also have picked $u = baba$ and $v = abba$.

Problem (Linear Temporal Logic Separation). Given finite sets P and N of lassos over an alphabet Σ, and a grammar $\mathcal{G}$ for LTL over Σ, synthesize a formula $\varphi \in L(\mathcal{G})$ such that $w \models \varphi$ for all $w \in P$ and $w \not\models \varphi$ for all $w \in N$, or declare no such formula exists.

Our LTL formulas come from the following grammar.

$$\varphi ::= a \in \Sigma \mid \varphi \wedge \varphi' \mid \varphi \vee \varphi' \mid \neg\varphi \mid \mathsf{X}\varphi \mid \varphi\mathsf{U}\varphi'$$

Below we present the semantics in a way that makes clear the aspects, which are *positions* in the lasso (u, v), along with an indication of whether the position corresponds to u or v. An LTL formula φ is true in a lasso (u, v), written $(u, v) \models \varphi$, precisely when $(u, v), (1, \_) \models \varphi$, with the latter defined below. The main idea is that the following holds

$$(u, v), (i, \_) \models \varphi \Leftrightarrow u^i v^\omega \models \varphi \quad \text{and} \quad (u, v), (\_, j) \models \varphi \Leftrightarrow v^j v^\omega \models \varphi,$$

where $uv^\omega \models \varphi$ is the standard semantics for LTL [Pnueli 1977]. To denote the letter at position i we write $w(i)$. We use i to range over $[1, |u|]$ and j to range over $[1, |v|]$. If $j' < j$, then $[j, j']$ means $[1, j'] \cup [j, |v|]$. We use $[a, b)$ to exclude b.

$(u, v), (i, \_)$	$\models$	$a \in \Sigma$	if	$u(i) = a$
$(u, v), (\_, j)$	$\models$	$a \in \Sigma$	if	$v(j) = a$
$(u, v), p$	$\models$	$\neg\varphi$	if	$(u, v), p \not\models \varphi$
$(u, v), p$	$\models$	$\varphi \wedge \varphi'$	if	$(u, v), p \models \varphi$ and $(u, v), p \models \varphi'$
$(u, v), p$	$\models$	$\varphi \vee \varphi'$	if	$(u, v), p \models \varphi$ or $(u, v), p \models \varphi'$
$(u, v), (|u|, \_)$	$\models$	$\mathsf{X}\varphi$	if	$(u, v), (\_, 1) \models \varphi$
$(u, v), (i, \_)$	$\models$	$\mathsf{X}\varphi$	if	$(u, v), (i + 1, \_) \models \varphi$ $i < |u|$
$(u, v), (\_, j)$	$\models$	$\mathsf{X}\varphi$	if	$(u, v), (\_, j \bmod |v| + 1) \models \varphi$
$(u, v), (i, \_)$	$\models$	$\varphi\mathsf{U}\varphi'$	if	$\exists i' \geq i.\ (u, v), (i', \_) \models \varphi'$ and $\forall i'' \in [i, i').\ (u, v), (i'', \_) \models \varphi$
			or	$\exists j.\ \forall i' \geq i.\ (u, v), (i', \_) \models \varphi$ and $\forall j' < j.\ (u, v), (\_, j') \models \varphi$
				and $(u, v), (\_, j) \models \varphi'$
$(u, v), (\_, j)$	$\models$	$\varphi\mathsf{U}\varphi'$	if	$\exists j'.\ \forall j'' \in [j, j').\ (u, v), (\_, j'') \models \varphi$ and $(u, v), (\_, j') \models \varphi'$

6.2 Decidable Learning for LTL

The FACET program LTL in Figure 6 reads LTL syntax trees and evaluates them over Σ-lassos, presented as pairs of finite words $(u, v) \in \Sigma^* \times \Sigma^*$. Again we omit *dual* clauses. The ranked alphabet Δ for syntax trees is similar to those of regular expressions and modal logic. Signature functions include functions for word length, written $|w|$, and functions for computing sets of consecutive positions, e.g., $[x, y]$ and $[x, y)$. There is also a function $wrap(j)$ defined by

$$wrap(j) = \text{if } j > |v| \text{ then } 1 \text{ else } j$$

which is used to reset the current lasso position to the beginning of the suffix v when it exceeds $|v|$. There are also functions for comparison of positions, e.g. $i < i'$, and equality and disequality for

```
LTL((u,v), (i,_), n) := match n.l with
    ∧ → LTL((u,v), (i,_), n.c₁) and LTL((u,v), (i,_), n.c₂)
    ∨ → LTL((u,v), (i,_), n.c₁) or LTL((u,v), (i,_), n.c₂)
    ¬ → LTL((u,v), dual(i,_), n.c₁)
    X → if i < |u| then LTL((u,v), (i+1,_), n.c₁) else LTL((u,v), (_,1), n.c₁)
    U → any (λi'. LTL((u,v), (i',_), n.c₂) and
                  all (λi''. LTL((u,v), (i'',_), n.c₁)) [i,i')) [i,|u|]
          or    all (λi'. LTL((u,v), (i',_), n.c₁)) [i,|u|] and
                  any (λj. all (λj'. LTL((u,v), (_,j'), n.c₁)) [1,j)
                                and LTL((u,v), (_,j), n.c₂)) [1,|v|]
    x → u(i) = x

LTL((u,v), (_,j), n) := match n.l with
    ∧ → LTL((u,v), (_,j), n.c₁) and LTL((u,v), (_,j), n.c₂)
    ∨ → LTL((u,v), (_,j), n.c₁) or LTL((u,v), (_,j), n.c₂)
    ¬ → LTL((u,v), dual(_,j), n.c₁)
    X → LTL((u,v), (_,wrap(j+1)), n.c₁)
    U → any (λj'. LTL((u,v), (_,j'), n.c₂) and
                  all (λj''. LTL((u,v), (_,j''), n.c₁)) [j,j')) [1,|v|]
    x → v(j) = x
```

Fig. 6. LTL evaluates the LTL formula φ pointed to by n over lasso (u,v) and verifies that $(u,v) \models \varphi$.

alphabet letters at a given position, e.g. $u(i) = x$. The states use two constructors $(\_, \cdot)$ and $(\cdot, \_)$ to encode whether a position is part of u or v. For a given lasso (u,v) the states are:

$$\text{Asp}((u,v)) := \{p, dual(p) \,:\, p \in \text{pos}\}$$
$$\text{pos} := \{(i,\_),(\_,j) \,:\, i \in [1,|u|], j \in [1,|v|]\}$$

THEOREM 6.1. *Linear temporal logic separation for finite sets P and N of lasso structures, and grammar $\mathcal{G}$, is decidable in time $\mathcal{O}(2^{poly(mn)} \cdot |\mathcal{G}|)$, with $m = |P| + |N|$ and $n = \max_{(u,v) \in P \cup N}(|uv|)$.*

PROOF SKETCH. For each lasso $(u,v) \in P$, positions $i \in [1,|u|]$, $j \in [1,|v|]$, and LTL formula φ, we have that $(u,v),(i,\_) \models \varphi$ if and only if $((u,v), root(\varphi), C(\text{LTL}), (i,\_)) \Downarrow_p$ and $(u,v),(\_,j) \models \varphi$ if and only if $((u,v), root(\varphi), C(\text{LTL}), (\_,j)) \Downarrow_p$. Similarly, for each lasso $(u,v) \in N$ we have that $(u,v),(i,\_) \not\models \varphi$ if and only if $((u,v), root(\varphi), C(\text{LTL}), dual((i,\_))) \Downarrow_p$ and $(u,v),(\_,j) \not\models \varphi$ if and only if $((u,v), root(\varphi), C(\text{LTL}), dual((\_,j))) \Downarrow_p$. The proof is by induction on φ. We have $|\text{Asp}((u,v))| = \mathcal{O}(|uv|)$ and the rest follows by Theorem 4.3 and Corollary 4.4. □
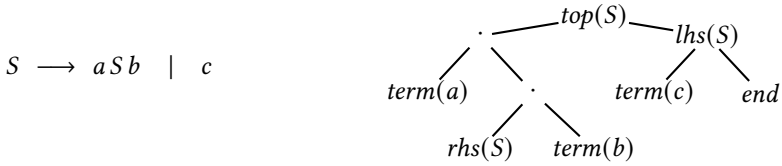
7 CONTEXT-FREE GRAMMARS

In this section we consider another problem involving separation of labeled finite words. The goal is to synthesize a context-free grammar that generates all positively-labeled words and no negatively-labeled words. We derive a decision procedure as before by writing a FACET program.

7.1 Separating Words with Context-Free Grammars

Problem (Context-Free Grammar Separation). Given finite sets P and N of finite words over an alphabet Σ, as well as a (meta-)grammar $\mathcal{G}$, synthesize a context-free grammar G over nonterminals

NT, terminals Σ, and axiom $S \in NT$, such that $G \in L(\mathcal{G})$ and $P \subseteq L(G)$ and $N \cap L(G) = \emptyset$, or declare no such grammar exists.

The semantics of context-free grammars (CFGs) is standard: a word is generated by a grammar if we can build a parse tree for it using the productions. We want to represent CFGs as syntax trees and then write a program that reads such trees and evaluates whether a given word is generated by the represented grammar. The syntax trees can organize productions along, say, the right spine, with their right-hand sides in left children as suggested below. Note that the ranked alphabet Δ uses a binary symbol $lhs(A)$ and nullary symbol $rhs(A)$ for each $A \in NT$ to distinguish between occurrences of A in the right-hand side of a production from occurrences in the left-hand side. We also use a binary symbol $top(S)$ to distinguish the root of the syntax tree, as well as a nullary symbol end to signal the end of productions along the right spine. Terminals $a \in \Sigma$ are represented as nullary symbols $term(a)$. See below with a grammar on the left and its syntax tree on the right.

$$S \longrightarrow a\,S\,b \quad | \quad c$$

7.2 Decidable Learning for Context-Free Grammars

We want a program that evaluates a CFG syntax tree G to verify whether $w \in L(G)$ for an input word w. What kind of state is needed? Intuition suggests the FACET evaluator will be similar to the one for regular expressions, and that we should use pairs of positions. The main difference is the more flexible recursion afforded by nonterminals. Consider reading the syntax tree above starting at the root labeled by $top(S)$, with the production "$a\,S\,b$" in the left subtree and the rest of the productions on the right. To verify that w is generated by S, the program should move to the right-hand sides of the two S-productions and check whether w is generated by *either* of these. Concatenation in the right-hand sides of productions can be handled just like for regular expressions by guessing a split for w and then verifying the guess in the subtrees. But upon reading, say, $rhs(S)$ in the production "$a\,S\,b$", the program should navigate *up* to find the S productions and reenter their right-hand sides in order to parse the current subword and verify its membership in $L(S)$. This is accomplished by entering a state $reset(S)$ that causes the program to navigate to the root of the syntax tree and then move downward in a state $find(S)$ to check membership in all productions for S. It turns out, however, that this more flexible recursion afforded by the nonterminals introduces a subtlety when verifying *non-membership* of a word in the grammar. We will return to this point after we consider the part of the program that verifies membership.

7.2.1 Verifying Membership. We write a program CFG that evaluates an input CFG syntax tree G over a word w and verifies that $w \in L(G)$. The states consist of ordered pairs of word positions as well as some extra information related to moving up and down on the syntax tree, along with duals:

$$\begin{aligned}
\mathrm{Asp}(w) &:= \{x, dual(x) \;:\; x \in X(w)\} \\
X(w) &:= \mathrm{subs}(w) \cup \{(s, find(A)), (s, reset(A)) \;:\; s \in \mathrm{subs}(w), A \in NT\} \\
\mathrm{subs}(w) &:= \{(l, r) \;:\; 1 \leq l \leq r \leq |w| + 1\}
\end{aligned}$$

Signature functions are the same as those for regular expressions. The clauses for CFG (duals omitted) are shown in Figure 7.

```
CFG(w, (l,r), n) := match n.l with
    · → any (λx. CFG(w, (l,x), n.c₁) and CFG(w, (x,r), n.c₂) [l,r]
    rhs(z) → CFG(w, (l,r), reset(z), n.up)
    term(x) → r = l + 1 and w(l) = x


CFG(w, (l,r), reset(z), n) := match n.l with
    top(z) → CFG(w, (l,r), n.c₁) or CFG(w, (l,r), find(z), n.c₂)
    top(x) → CFG(w, (l,r), find(z), n.c₂)
    _      → CFG(w, (l,r), reset(z), n.up)


CFG(w, (l,r), find(z), n) := match n.l with
    lhs(z) → CFG(w, (l,r), n.c₁) or CFG(w, (l,r), find(z), n.c₂)
    lhs(x) → CFG(w, (l,r), find(z), n.c₂)
```

Fig. 7. CFG evaluates an input CFG syntax tree G pointed to by n against word w and verifies that $w \in L(G)$.

Consider the operation of CFG over a word w and a grammar G that has a production like $A \to AA$. Notice that the program could read this production arbitrarily many times in the same state by always choosing to split w into ϵ and w when reading the right-hand side AA, in which case it would verify recursively that $w \in L(A)$ and $\epsilon \in L(A)$. Nevertheless, if indeed $w \in L(G)$, then there is a finite proof for $(w, root(G), C(\text{CFG}), ((1, |w| + 1), reset(S))) \Downarrow_p$ that can be obtained by following any correct derivation of w from the grammar. The case for $w \notin L(G)$ is more subtle.

7.2.2 Verifying Non-Membership. Consider how CFG should check $w \notin L(A)$ for a production like $A \to AA$. Now there is no derivation to follow, and the program might loop forever by entering the right-hand side and reading AA, which will cause it to read all A productions, which will cause it to read AA, and so on, with no guarantee that subwords become smaller in each recursive call. This termination issue can be dealt with in a few ways, e.g., by adding states to keep track of the depth of recursion, but we present a simpler solution that avoids this.

It turns out that the duals for the CFG clauses in Figure 7 are sufficient for our purposes, provided all input grammars are in *Greibach normal form* (GNF). Productions in GNF grammars have the form $A \to a(NT)^*$, with $a \in \Sigma$ and $A \in NT$. Intuitively, this restriction helps because it makes proofs of non-membership finite: subwords must become smaller each time the program recursively checks a given nonterminal. To see this, consider verifying $aba \notin L(S)$ for the GNF grammar:

$$S \longrightarrow a\,S \mid b$$

The word aba is clearly not generated by the second production. To show it is not generated by the first, we show there is no way to split aba into $w_1 w_2$ so that w_1 is generated by a and w_2 is generated by S. If $w_1 \neq a$ then the subproof for that split can end. Otherwise $w_1 = a$, and thus $|w_2| < w$, and hence the subproof for $w_2 \notin L(S)$ will be finite by induction on word length. For any GNF grammar G and word $w \notin L(G)$, there is a finite proof for $(w, root(G), C(\text{CFG}), dual((1, |w|+1), reset(S))) \Downarrow_p$, and the argument does not in fact rely on the precise form of GNF productions; it works for more general productions of the form $A \to \alpha$ with $\alpha \in (\Sigma \sqcup NT)^* \Sigma (\Sigma \sqcup NT)^*$, i.e., those that involve at least one terminal. We call a grammar *productive* if each of its productions meets this requirement. As long as input grammars are productive, the dual clauses for those from Figure 7 correctly verify

non-membership. Note that all context-free languages can be represented by productive CFGs, and thus we assume that the input meta-grammar $\mathcal{G}$ encodes only productive CFGs[6].

THEOREM 7.1. *CFG separation for finite sets of words P, N, and grammar $\mathcal{G}$ enforcing productivity, is decidable in time $\mathcal{O}(2^{poly(k)} \cdot |\mathcal{G}|)$, where $n = \max_{w \in P \cup N} |w|$, $m = |P| + |N|$, and $k = mn^2 \cdot |NT|$.*

PROOF SKETCH. Fix a word w. For every $(i, j) \in \text{subs}(w)$ and for every G we have that $w(i, j) \in L(G)$ if and only if $(w, root(G), C(\text{CFG}), ((i, j), reset(S))) \Downarrow_p$. Similarly, we have $w(i, j) \notin L(G)$ if and only if $(w, root(G), C(\text{CFG}), dual((i, j), reset(S))) \Downarrow_p$. The proof is by induction on w and G. We have $|\text{Asp}(w)| = \mathcal{O}(|w|^2 \cdot |NT|)$ and the theorem follows by Theorem 4.3 and Corollary 4.4. □

8 FIRST-ORDER LOGIC OVER RATIONAL NUMBERS WITH ORDER

In this section we consider learning first-order logic queries over an infinite domain, namely, the structure $(\mathbb{Q}, <)$ consisting of the rational numbers $\mathbb{Q}$ with the usual linear order $<$. The learning problem requires labeled k-tuples of rational numbers to be separated by a *query* in FO^k, i.e., a formula in first-order logic with k variables. We derive a decision procedure by writing a FACET interpreter for FO^k over $(\mathbb{Q}, <)$.

8.1 Learning Queries over Rational Numbers with Order

We consider the following problem.

Problem (Learning FO^k Queries over $(\mathbb{Q}, <)$). Given finite sets P and N of k-tuples of $\mathbb{Q}$, and grammar $\mathcal{G}$ for FO^k over $(\mathbb{Q}, <)$, synthesize $\varphi \in L(\mathcal{G})$ such that $P \subseteq \{t \in \mathbb{Q}^k \mid (\mathbb{Q}, <) \models \varphi(t)\}$ and $N \subseteq \{t \in \mathbb{Q}^k \mid (\mathbb{Q}, <) \not\models \varphi(t)\}$, or declare no such formula exists.

A ranked alphabet Δ for FO^k has, for any variables x, y, the unary symbols "$\forall x$", "$\exists x$" and nullary symbols "$x < y$", "$x = y$", in addition to the symbols for Boolean operators. Note that for this problem the *language* takes the class of structures $\mathcal{M}$ to be the set $\mathbb{Q}^k$, and so a single "structure" is a tuple of rationals $t \in \mathbb{Q}^k$. For $t \in \mathbb{Q}^k$, the semantics is given by $t \models \varphi \Leftrightarrow (\mathbb{Q}, <), t \models \varphi$[7].

8.2 Decidable Learning for First-Order Logic Queries over Rationals

Given $t \in \mathbb{Q}^k$ and $\varphi \in \text{FO}^k$, what semantic information do we need to verify $(\mathbb{Q}, <) \models \varphi(t)$? Consider evaluating a query $\varphi(x, y, z)$ on $t = (1/2, 3, 4/3) \in P$. Our program might begin with an assignment γ mapping (x, y, z) to $(1/2, 3, 4/3)$. With the right functions, it can easily verify atomic formulas by simply checking $\gamma(x) < \gamma(y)$ or $\gamma(x) = \gamma(y)$. When the program reads, say, "$\exists x$", it must carry forward some finite amount of information, which thus excludes tracking the precise values for the variables, of which there are infinitely many.

The main idea is that evaluating atomic formulas does not require the precise values of the variables: the *order between variables* is all that is needed to evaluate FO^k formulas over $(\mathbb{Q}, <)$. In our example, we have $t = (1/2, 3, 4/3)$ corresponding to the assignment $\{x \mapsto 1/2, y \mapsto 3, z \mapsto 4/3\}$, and so the program begins in a state encoding that $x < z < y$. Suppose it reads the formula $\exists x \forall y \, (x < y)$. First it reads "$\exists x$" and branches (disjunctively) on all of the finitely-many *distinguishable* choices for where to place x relative to the other variables while leaving the others in the same relative positions. We preserve $z < y$, but x can appear in any of several positions: $x < z < y$, $x = z < y$, $z < x < y$, $z < x = y$, $z < y < x$. From each of these states the program reads "$\forall y$" and branches (conjunctively) on every choice for where to place y. It eventually rejects the formula because y can always be placed strictly below x in the second branching step.

---

[6]Alternatively, we can use another automaton to verify that input trees encode productive grammars. The product of this automaton with the meta-grammar automaton $\mathcal{A}_{\mathcal{G}}$ can itself be viewed as a meta-grammar which enforces productivity.

[7]We are abusing notation and treating t, a k-tuple of rationals, as an assignment to the k, ordered free variables of φ.

```
Rat(t, ≳, n) := match n.l with
    ∀x → all (λp. Rat(t, p, n.c₁)) place(x,≳)
    ∃x → any (λp. Rat(t, p, n.c₁)) place(x,≳)
    ∧ → Rat(t, ≳, n.c₁) and Rat(t, ≳, n.c₂)
    ∨ → Rat(t, ≳, n.c₁) or Rat(t, ≳, n.c₂)
    ¬ → Rat(t, dual(≳), n.c₁)
    x < z → if lt(x,z,≳) then True else False
    x = z → if eq(x,z,≳) then True else False
```

Fig. 8. Rat evaluates formula φ pointed to by n against a tuple t of rational numbers and verifies $(\mathbb{Q}, <) \models \varphi(t)$.

Figure 8 shows a program Rat that evaluates FO^k formulas over tuples of rational numbers. The states of Rat record an ordering between variables $V = \{y_1, \ldots, y_k\}$, including whether two variables are equal, and thus they correspond to the *total preorders* on V, denoted $pre(V)$. We use "$\gtrsim$" as a pattern variable to denote a preorder. For a given tuple t we have

$$\text{Asp}(t) := \{ \gtrsim, dual(\gtrsim) : \gtrsim \in pre(V) \} .$$

Signature functions include Boolean functions for checking ordering and equality in a given preorder $\gtrsim$, which we denote by $lt(x, z, \gtrsim)$ and $eq(x, z, \gtrsim)$ and define by:

$$
\begin{aligned}
geq(x, z, \gtrsim) &:= & x \gtrsim z & \qquad & eq(x, z, \gtrsim) &:= & x \gtrsim z \text{ and } z \gtrsim x \\
lt(x, z, \gtrsim) &:= & \neg geq(x, z, \gtrsim) & \qquad & neq(x, z, \gtrsim) &:= & \neg eq(x, z, \gtrsim)
\end{aligned}
$$

State functions include $place(x, \gtrsim)$, which computes the set of all total preorders that place $x \in V$ in a new position but agree with $\gtrsim$ on variables in $V \setminus \{x\}$, defined as:

$$place(x, \gtrsim) := \{ \gtrsim' \in pre(V) : y \gtrsim' z \Leftrightarrow y \gtrsim z, \forall y, z \in V \setminus \{x\} \} .$$

THEOREM 8.1. *Learning queries over rationals with order, with sets of k-tuples P and N and grammar $\mathcal{G}$ for FO^k, is decidable in time $\mathcal{O}(2^{poly(mk^k)} \cdot |\mathcal{G}|)$, where $m = |P| + |N|$.*

PROOF SKETCH. Follows reasoning from previous sections and uses Theorem 4.3 and Corollary 4.4. For $t \in \mathbb{Q}^k$ and a set V of k variables, we have $|\text{Asp}(t)| = 2|pre(V)| = \mathcal{O}(k^k)$.

□

Remark. Structures like $(\mathbb{Q}, <)$ have a special kind of automorphism group, called an *oligomorphic group* (see [Hodges 1993]). Oligomorphic automorphism groups have finitely-many orbits in their action on n-tuples from the domain of the structure, for every n. For structures with such automorphism groups, if n is fixed, then a FACET program can evaluate formulas by tracking these finitely-many orbits. In the case of FO^k over $(\mathbb{Q}, <)$, the program Rat effectively checks atomic formulas in a given orbit represented by a total preorder on variables, and when evaluating quantifiers it is able to compute all "nearby" orbits. There are other examples of such structures, and many constraint satisfaction problems over infinite domains use them as templates, e.g., temporal constraint satisfaction, phylogenetic reconstruction over tree structures, and network satisfaction problems (see [Bodirsky 2021]). It would be interesting to explore learning in domains like these.

9 DECIDABLE LEARNING FOR STRING PROGRAMS

In this section, we consider Gulwani's language for string programming [Gulwani 2011], which is designed to express programs that transform a sequence i of input strings into an output string o

in the context of spreadsheet programs. This language, which we refer to as String, turns out to be FAC, with the caveat that loops must use variables from a finite set. We next give an overview of the syntax and semantics of String; details can be found in the original paper [Gulwani 2011]. Then we discuss how to implement a Facet evaluator that reads String syntax trees and checks whether they map an input sequence i to an output o. The language, being one used in practice, is considerably more complex than our other examples, and so we only sketch the main ideas.

9.1 String Overview

Programs in String map finitely-many input strings v_j to an output o. A *program* $P \in$ String consists of a switch statement Switch$((\varphi_1, e_1), \ldots, (\varphi_n, e_n))$ that chooses between expressions e_i. The φ_i are DNF formulas over atoms Match(v_j, r, k), which hold if at least k matches for a regular expression r can be found in the input v_j. The e_i in the switch statement, called *trace expressions*, have the form Concatenate$(f_1, \ldots, f_n)$. They concatenate expressions f_i that come in three flavors: (1) SubStr(v_j, p_1, p_2) selects the substring in v_j between positions p_1 and p_2, (2) ConstStr(s) denotes a string literal s, and (3) Loop$(\lambda x. e)$ iteratively appends the result of evaluating e until that result is $\bot$, which is a special value for failure. During iteration i, the variable x is bound to i in e.

The positions p_i in SubStr(v_j, p_1, p_2) are either constant integers CPos(k) or they have the form Pos(r_1, r_2, c), where the r_i are regular expressions and c is a linear integer expression built from constants and loop variables, e.g. $2x + 3$. The expression Pos(r_1, r_2, c) is evaluated with respect to v_j, and returns a position t such that just to the left of t in v_j there is a match for r_1 and starting at t there is a match for r_2. Furthermore, it returns the cth such position, or $\bot$ if not enough such positions exist. Regular expressions are restricted in String to only use Kleene star and disjunction in a particular way, which we ignore. It is no trouble to write a Facet evaluator for a generalization of String that allows unrestricted (extended) regular expressions like those from Section 5.

Consider a program that extracts capital letters of an input string ([Gulwani 2011], example 5).

Input v_1	Output o
Principles Of Programming Languages	*POPL*

Program: Loop$(\lambda x.$ Concatenate$($SubStr2$(v_1,$ UpperTok, $x)))$

where   SubStr2$(v_j, r, c) \equiv$ SubStr$(v_j,$ Pos$(\epsilon, r, c),$ Pos$(r, \epsilon, c))$

This program uses SubStr2(v_j, r, c) to compute the cth match of the regular expression r in v_j. This is used to extract the xth upper case letter in iteration x of the loop, which is then appended to previously extracted letters. The loop exits when the body evaluates to $\bot$, which happens when there are no more matches for UpperTok.

9.2 Decidable Learning for String

We describe how a Facet program should evaluate the constructs in String. Fix a set of input strings $i = v_1, \ldots, v_n$ and an output string o. The evaluator reads $P \in$ String and checks that it maps i to o. We mention specific choices for representing syntax trees as needed.

The Switch statement can be modeled with a ternary symbol Switch$(\varphi, e, \text{rest})$, where rest represents the rest of the cases with nested operators of the same kind. Upon reading Switch, the program branches to verify *either* the conditional φ_1 holds and e_1 produces o or φ_1 does not hold and the rest of the Switch produces o.

The DNF formulae φ_i can be easily evaluated with the Boolean operators in Facet. An atom Match(v_i, r, k) can be represented with a binary symbol Match$_{v_i}(r, k)$, one for each v_i, with right child a *unary* representation of integer k, i.e. $s^k(0)$. To check Match$_{v_i}(r, s^k(0))$, the program evaluates the right child to determine the value of k. Crucially, it can reject if k exceeds $|\text{subs}(v_i)|$,

which upper bounds the maximum number of matches for any regular expression over v_i. Having determined k, the program can branch over all $\binom{N}{k}$ combinations of subwords that could witness the requisite k matches, with $N = |\text{subs}(v_i)|$. For each subword, the program executes Reg (Figure 5) as a subroutine to check whether it matches the regular expression in the left child.

It remains to interpret e and verify it produces an output o. We represent $\text{Concatenate}(f_1, \ldots, f_n)$ in a nested way like Switch, and binary $\text{Concatenate}(f, f')$ is evaluated as for regular expressions by branching on all ways to split o (or one of its subwords) into consecutive subwords w and w', with f and f' then verified to produce w and w'.

The expressions f are verified to yield a given word as follows. Literals $\text{ConstStr}(s)$ are represented with nested concatenation and thus follow the same idea as $\text{Concatenate}(f, f')$. Substrings $\text{SubStr}(v_j, p_1, p_2)$ are modeled with binary symbols $\text{SubStr}_{v_j}(p_1, p_2)$, one for each v_j. Having determined the values of positions p_i, the program can simply use a function for equality of subwords. Constant positions $\text{CPos}(k)$ are determined as before except the program rejects if k exceeds $|v_j|$. To evaluate $\text{Pos}(r_1, r_2, c)$, represented as a ternary operator, the program guesses a position t in v_j and verifies existence of matches for r_1 and r_2 to the left and right of t. It further verifies there are $c - 1$, *but not c* such positions to the left of t. This is accomplished by branching on the possible $\binom{t-1}{c-1}$ combinations of positions and checking for the requisite matches, and then checking the opposite for each of the $\binom{t-1}{c}$ combinations. Finally, integer expressions $c = k_1 x + k_2$ can be evaluated by hardcoding rules for bounded arithmetic, because the maximum value that loop variables can take is bounded by $|o|$, which we discuss next.

Provided the number of loop variables is finite, the program can evaluate loops using a map γ from variables $\{x_i\}$ to integers. The integers are bounded because the loop body e must produce a string of non-zero length (otherwise the loop terminates), and loop expressions are only ever verified to produce words of length no more than $|o|$. Since each iteration must productively decompose a word of length bounded by $|o|$, we can use $|o|$ as a bound on the range of γ. Thus the γ have finite domain and range and require finitely-many states. Now, suppose the program encounters a loop $\text{Loop}(\lambda x. e)$ with current variable map γ, and suppose it must verify the loop produces a word w. It first sets $\gamma(x) = 1$. Then it guesses a decomposition of w into $w_1 w_2$ such that e evaluated with γ produces w_1 and $\text{Loop}(\lambda x. e)$ evaluated with $\gamma[x \mapsto \gamma(x) + 1]$ produces w_2.

We conclude by Theorem 4.3 that learning STRING programs from examples is decidable, even for the generalization that allows unrestricted regular expressions (which [Gulwani 2011] disallows).

10 RELATED WORK

Expression Learning and Program Synthesis. Our approach is inspired by recent results for learning in finite-variable logics [Krogmeier and Madhusudan 2022]. Proofs in that work involve direct automata constructions, and the results can be obtained with our meta-theorem by writing suitable evaluators. Our work generalizes the tree automata approach to general symbolic languages by separating decidable learning theorems into two parts: (1) identifying the underlying semantic aspects of the language in question and (2) programming with this new datatype in order to evaluate arbitrary expressions. The finite-variable restriction for the logics considered in [Krogmeier and Madhusudan 2022] leads to finitely-many aspects— a finite set of assignments to some k variables. But, as our work shows, this restriction is not necessary for decidable learning; several languages we consider do not use variables and it is unclear what a corresponding variable restriction would mean. Usual translations of regular expressions to monadic second-order logic formulae, for instance, do not stay within a finite-variable fragment. Nevertheless, the recursive semantics of regular expressions involves subwords, and there are only finitely-many subwords of a given word, which makes regular expressions finite-aspect checkable.

Practical algorithms for some of the learning problems we address have been explored previously, e.g. learning LTL [Neider and Gavran 2018], regular expressions [Fernau 2009; Li et al. 2008], and context-free grammars [Langley and Stromsten 2000; Sakakibara 2005; Vanlehn and Ball 1987], but decidable learning results with syntactic restrictions have not been established.

Other recent work studies the parameterized complexity of learning queries in FO [van Bergerem et al. 2022], algorithms for learning in FO with counting [van Bergerem 2019], and learning in description logics [Funk et al. 2019]. Applications for FO learning have emerged, e.g., synthesizing invariants [Garg et al. 2014, 2015; Hance et al. 2021; Koenig et al. 2020, 2022; Yao et al. 2021] and learning program properties [Astorga et al. 2019, 2021; Miltner et al. 2020].

Expression learning is connected to program synthesis, and in particular, programming by example [Polozov and Gulwani 2015], where practical algorithms have been used to automate tedious programming tasks, e.g. synthesizing string programs [Cambronero et al. 2023; Gulwani 2011], bit-manipulating programs from templates [Solar-Lezama et al. 2006], or functional programs from examples and type information [Osera and Zdancewic 2015; Polikarpova et al. 2016]. Synthesis with grammar restrictions follows work in the SyGuS [Alur et al. 2015], and recently, SemGuS frameworks [Kim et al. 2021].

Automata for Synthesis. Connections between automata and synthesis go back to Church's problem [Church 1963] on synthesizing finite state machines that manipulate infinite streams of bits to meet a given logical specification. This was solved first by Büchi and Landweber [Buchi and Landweber 1969] for specifications in monadic second-order logic, and later also by Rabin [Rabin 1972]. The idea was to translate the specification into an automaton, and to view synthesis of a transducer as the problem of synthesizing a finite-state winning strategy in a game played on the transition graph of the automaton. The result was a potentially large transition system, not a compact program. The use of tree automata that work over syntax trees was advanced in [Madhusudan 2011] and has been used for practical algorithms in several program synthesis contexts [Handa and Rinard 2020; Koppel et al. 2022; Miltner et al. 2022; Wang et al. 2017a,b, 2018].

Decidability in Synthesis. Many foundational decidability results in logic and synthesis of finite-state systems rely on reductions to automata emptiness [Buchi and Landweber 1969; Grädel et al. 2002; Kupferman et al. 2000, 2010; Pnueli and Rosner 1989, 1990; Rabin 1972]. Recent decidability results for synthesis of uninterpreted programs involved a reduction to emptiness of two-way tree automata [Krogmeier et al. 2020]. Decision procedures for SyGuS problems in linear and conditional linear integer arithmetic [Farzan et al. 2022; Hu et al. 2020] used grammar flow analysis [Möncke and Wilhelm 1991] and an abstraction based on semi-linear sets.

Automata for Learning vs. Graph Algorithms. There is a large body of work (e.g. see [Courcelle and Engelfriet 2012; Habel 1992]) on checking properties of graphs expressed in monadic second-order logic. These results involve translating logical properties into automata that read *decompositions of graphs* and accept if the represented graph has the property. Our work is very differently motivated: we are interested in properties of syntax trees defined over arbitrary *fixed* structures (e.g. unrestricted graphs like cliques or grids), and the properties are motivated by semantics of complex symbolic languages. Our automata constructions for *learning* are, conceptually, dual to these constructions from logical specifications.

Definability in Monadic Second-Order Logic. Foundational results from logic and automata theory connect definability in monadic second-order logic and recognizability by finite machines, spanning various classes of structures including finite words and trees [Büchi 1960; Doner 1970; Elgot 1961; Thatcher and Wright 1968; Trakhtenbrot 1961], infinite words and trees [Büchi 1990; Rabin 1969], and graphs with bounded tree width [Courcelle 1990]. It follows by definition that for any FAC language, the semantics over any fixed structure can be captured by a sentence in monadic-second order logic over syntax trees.

11 CONCLUSION

We introduced a powerful recipe for proving that a symbolic language has decidable learning. It involves writing a program, i.e. semantic evaluator, that operates over mathematical structures and expression syntax trees for a given symbolic language. *Finite-aspect checkable* languages have the property that the semantics of any expression e can be expressed in terms of a finite amount of semantic information (aspects) that depends on the structure over which evaluation occurs but not on the size of e. This addresses a central question in expression learning with *version space algebra* (VSA) techniques, especially those realized as tree automata: for which symbolic languages are these learning algorithms possible? One prevailing answer in the literature is that language operators should have *finite inverses* (e.g., see [Cambronero et al. 2023; Polozov and Gulwani 2015]). When operators have finite inverses a top-down tree automaton can be effectively constructed. Our work suggests a weaker requirement, namely, that it be sufficient to *evaluate* arbitrarily large expressions or programs on specific examples by traversing syntax trees up and down using memory that is bounded by a function solely of the *size of the example*. For instance, in Section 8 we considered learning queries over the rational numbers with order. The "inverse" of $x < y$ is an infinite set of ordered pairs of rational numbers. Nevertheless, to evaluate a formula for a specific example, all that is needed is a bounded number of bits to encode the current ordering of variables.

We have also presented a set of interesting FAC languages that have nontrivial semantic definitions using finitely-many aspects, and new decidable learning results for each. We believe that many more can be readily found using our meta-theorem.

Tree automata underlie many practical algorithms for synthesis based on compactly representing large spaces of programs and expressions [Gulwani 2011; Handa and Rinard 2020; Koppel et al. 2022; Miltner et al. 2022; Wang et al. 2017a,b, 2018]. The main idea is to efficiently represent classes of expressions which are equivalent with respect to some examples. This idea originates with version space algebra [Mitchell 1982, 1997], which essentially amounts to a restricted form of tree automata working over trees of bounded depth [Koppel 2021]. Bringing the full tree automata toolkit to bear on learning and synthesis, e.g. two-way power and alternation, recognizes tree automata as a kind of basic building block for a version space algebra *over tree automata*. The learning constructions from our work use semantic evaluators (compact, effective descriptions of tree automata) as a basic building block and combine them in specific ways to address specific learning problems. Given this uniform technique of using tree automata as a programming language, it would be interesting to build compact representations and incremental algorithms for their construction and emptiness that yield generic learning algorithms which scale. Bounding the depth of expressions may make some of the constructions from this paper feasible.

Tree automata have also been used in many other contexts in computer science. In fixed-parameter tractable algorithms (e.g. Courcelle's theorem [Flum and Grohe 2006]) and finite model theory, they have been used to obtain generic algorithms for MSO-definable properties that work over tree decompositions of graphs, while in temporal logic verification [Esparza et al. 2021] they have been used as acceptors of correct behaviors of systems. Their use for learning in symbolic languages is an emerging new application of tree automata. It would be interesting to study the *theory* of FAC languages in terms of expressiveness, language-theoretic properties, and alternative characterizations.

REFERENCES

Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2015. Syntax-Guided Synthesis. In *Dependable Software Systems Engineering*. NATO Science for Peace and Security Series, D: Information and Communication Security, Vol. 40. IOS Press, 1–25.

Angello Astorga, P. Madhusudan, Shambwaditya Saha, Shiyu Wang, and Tao Xie. 2019. Learning Stateful Preconditions modulo a Test Generator. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 775–787. https://doi.org/10.1145/3314221.3314641

Angello Astorga, Shambwaditya Saha, Ahmad Dinkins, Felicia Wang, P. Madhusudan, and Tao Xie. 2021. Synthesizing Contracts Correct modulo a Test Generator. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 104 (oct 2021), 27 pages. https://doi.org/10.1145/3485481

Dietmar Berwanger. 2003. Game Logic is Strong Enough for Parity Games. *Studia Logica* 75, 2 (01 Nov 2003), 205–219. https://doi.org/10.1023/A:1027358927272

Patrick Blackburn, Maarten de Rijke, and Yde Venema. 2001. *Modal Logic*. Cambridge University Press. https://doi.org/10.1017/CBO9781107050884

Manuel Bodirsky. 2021. *Complexity of Infinite-Domain Constraint Satisfaction*. Cambridge University Press. https://doi.org/10.1017/9781107337534

J.C. Bradfield. 1998. The modal mu-calculus alternation hierarchy is strict. *Theoretical Computer Science* 195, 2 (1998), 133–153. https://doi.org/10.1016/S0304-3975(97)00217-X Concurrency Theory.

J. Richard Büchi. 1990. *On a Decision Method in Restricted Second Order Arithmetic*. Springer New York, New York, NY, 425–435. https://doi.org/10.1007/978-1-4613-8928-6_23

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

J. Richard Büchi. 1960. Weak Second-Order Arithmetic and Finite Automata. *Mathematical Logic Quarterly* 6, 1-6 (1960), 66–92. https://doi.org/10.1002/malq.19600060105 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/malq.19600060105

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

José Cambronero, Sumit Gulwani, Vu Le, Daniel Perelman, Arjun Radhakrishna, Clint Simon, and Ashish Tiwari. 2023. FlashFill++: Scaling Programming by Example by Cutting to the Chase. In *Principles of Programming Languages*. ACM SIGPLAN, ACM. https://www.microsoft.com/en-us/research/publication/flashfill-scaling-programming-by-example-by-cutting-to-the-chase/

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

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

Bruno Courcelle. 1990. The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. *Information and Computation* 85, 1 (1990), 12–75. https://doi.org/10.1016/0890-5401(90)90043-H

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

John Doner. 1970. Tree acceptors and some of their applications. *J. Comput. System Sci.* 4, 5 (1970), 406–451. https://doi.org/10.1016/S0022-0000(70)80041-1

Calvin C. Elgot. 1961. Decision Problems of Finite Automata Design and Related Arithmetics. *Trans. Amer. Math. Soc.* 98, 1 (1961), 21–51. http://www.jstor.org/stable/1993511

Javier Esparza, Orna Kupferman, and Moshe Y. Vardi. 2021. Verification. In *Handbook of Automata Theory*, Jean-Éric Pin (Ed.). European Mathematical Society Publishing House, Zürich, Switzerland, 1415–1456. https://doi.org/10.4171/Automata-2/16

Richard Evans and Edward Grefenstette. 2018. Learning Explanatory Rules from Noisy Data. *J. Artif. Int. Res.* 61, 1 (Jan. 2018), 1–64.

Azadeh Farzan, Danya Lette, and Victor Nicolet. 2022. Recursion Synthesis with Unrealizability Witnesses. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 244–259. https://doi.org/10.1145/3519939.3523726

Henning Fernau. 2009. Algorithms for learning regular expressions from positive data. *Information and Computation* 207, 4 (2009), 521–541. https://doi.org/10.1016/j.ic.2008.12.008

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

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. 2014. ICE: A Robust Framework for Learning Invariants. In Computer Aided Verification, Armin Biere and Roderick Bloem (Eds.). Springer International Publishing, Cham, 69–87.

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.

Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-Output Examples. In Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11). Association for Computing Machinery, New York, NY, USA, 317–330. https://doi.org/10.1145/1926385.1926423

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

Travis Hance, Marijn Heule, Ruben Martins, and Bryan Parno. 2021. Finding Invariants of Distributed Systems: It's a Small (Enough) World After All. In 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21). USENIX Association, 115–131. https://www.usenix.org/conference/nsdi21/presentation/hance

Shivam Handa and Martin C. Rinard. 2020. Inductive Program Synthesis over Noisy Data. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020). Association for Computing Machinery, New York, NY, USA, 87–98. https://doi.org/10.1145/3368089.3409732

Wilfrid Hodges. 1993. The countable case. Cambridge University Press, 323–359. https://doi.org/10.1017/CBO9780511551574.009

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 (PLDI 2020). Association for Computing Machinery, New York, NY, USA, 1128–1142. https://doi.org/10.1145/3385412.3385979

Radoslav Ivanov, Kishor Jothimurugan, Steve Hsu, Shaan Vaidya, Rajeev Alur, and Osbert Bastani. 2021. Compositional Learning and Verification of Neural Network Controllers. ACM Trans. Embed. Comput. Syst. 20, 5s, Article 92 (sep 2021), 26 pages. https://doi.org/10.1145/3477023

Michael J. Kearns and Umesh Vazirani. 1994. An Introduction to Computational Learning Theory. The MIT Press. https://doi.org/10.7551/mitpress/3897.001.0001

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

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 (PLDI 2020). Association for Computing Machinery, New York, NY, USA, 703–717. https://doi.org/10.1145/3385412.3386018

Jason R. Koenig, Oded Padon, Sharon Shoham, and Alex Aiken. 2022. Inferring Invariants with Quantifier Alternations: Taming the Search Space Explosion. In Tools and Algorithms for the Construction and Analysis of Systems, Dana Fisman and Grigore Rosu (Eds.). Springer International Publishing, Cham, 338–356.

James Koppel. 2021. Version Space Algebras are Acyclic Tree Automata. https://doi.org/10.48550/ARXIV.2107.12568

James Koppel, Zheng Guo, Edsko de Vries, Armando Solar-Lezama, and Nadia Polikarpova. 2022. Searching Entangled Program Spaces. Proc. ACM Program. Lang. 6, ICFP, Article 91 (aug 2022), 29 pages. https://doi.org/10.1145/3547622

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

Paul Krogmeier, 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.

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.

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

Pat Langley and Sean Stromsten. 2000. Learning Context-Free Grammars with a Simplicity Bias. In Machine Learning: ECML 2000, Ramon López de Mántaras and Enric Plaza (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 220–228.

Yunyao Li, Rajasekar Krishnamurthy, Sriram Raghavan, Shivakumar Vaithyanathan, and H. V. Jagadish. 2008. Regular Expression Learning for Information Extraction. In EMNLP.

P. Madhusudan. 2011. Synthesizing Reactive Programs. In Computer Science Logic (CSL'11) - 25th International Workshop/20th Annual Conference of the EACSL (Leibniz International Proceedings in Informatics (LIPIcs)), Marc Bezem (Ed.), Vol. 12. Schloss

Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 428–442. https://doi.org/10.4230/LIPIcs.CSL.2011.428

Kenneth L. McMillan. 1992. *Symbolic Model Checking: an approach to the state explosion problem.* Ph.D. Dissertation. Carnegie Mellon. thesis.pdf CMU Tech Rpt. CMU-CS-92-131.

Anders Miltner, Adrian Trejo Nuñez, Ana Brendel, Swarat Chaudhuri, and Isil Dillig. 2022. Bottom-up Synthesis of Recursive Functional Programs Using Angelic Execution. *Proc. ACM Program. Lang.* 6, POPL, Article 21 (jan 2022), 29 pages. https://doi.org/10.1145/3498682

Anders Miltner, Saswat Padhi, Todd Millstein, and David Walker. 2020. Data-Driven Inference of Representation Invariants. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020).* Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3385412.3385967

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.

Ulrich Möncke and Reinhard Wilhelm. 1991. Grammar flow analysis. In *Attribute Grammars, Applications and Systems*, Henk Alblas and Bořivoj Melichar (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 151–186.

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

Daniel Neider, P. Madhusudan, Shambwaditya Saha, Pranav Garg, and Daejun Park. 2020. A Learning-Based Approach to Synthesizing Invariants for Incomplete Verification Engines. *Journal of Automated Reasoning* 64, 7 (01 Oct 2020), 1523–1552. https://doi.org/10.1007/s10817-020-09570-z

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 (PLDI '15).* Association for Computing Machinery, New York, NY, USA, 619–630. https://doi.org/10.1145/2737924.2738007

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

Amir Pnueli and Roni Rosner. 1989. On the Synthesis of a Reactive Module. In *POPL.* ACM Press, 179–190.

Amir Pnueli and Roni Rosner. 1990. Distributed Reactive Systems Are Hard to Synthesize. In *FOCS.* IEEE Computer Society, 746–757.

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

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

Michael Oser Rabin. 1972. *Automata on Infinite Objects and Church's Problem.* American Mathematical Society, Boston, MA, USA.

Yasubumi Sakakibara. 2005. Learning context-free grammars using tabular representations. *Pattern Recognition* 38, 9 (2005), 1372–1383. https://doi.org/10.1016/j.patcog.2004.03.021 Grammatical Inference.

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 (ASPLOS XII).* Association for Computing Machinery, New York, NY, USA, 404–415. https://doi.org/10.1145/1168857.1168907

James W. Thatcher and Jesse B. Wright. 1968. Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical systems theory* 2 (1968), 57–81. https://doi.org/10.1007/BF01691346

Boris A. Trakhtenbrot. 1961. Finite automata and logic of monadic predicates. *Doklady Akademii Nauk SSSR* 140, 326-329 (1961), 122–123.

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. 2022. On the Parameterized Complexity of Learning First-Order Logic. In *Proceedings of the 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS '22).* Association for Computing Machinery, New York, NY, USA, 337–346. https://doi.org/10.1145/3517804.3524151

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

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

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017a. 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. 2017b. 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

Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. 2021. DistAI: Data-Driven Automated Invariant Learning for Distributed Protocols. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 405–421. https://www.usenix.org/conference/osdi21/presentation/yao

He Zhu, Stephen Magill, and Suresh Jagannathan. 2018. A Data-Driven CHC Solver. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 707–721. https://doi.org/10.1145/3192366.3192416

A   DETAILED DESCRIPTION OF Facet

Programs in Facet are parameterized by a language $(\mathcal{L}, \mathcal{M}, \models)$, where the semantic function $(\_ \models \_) : \mathcal{L} \times \mathcal{M} \to D$ specifies the domain D in which expressions are interpreted. A program takes as input a *pointer* into the syntax tree for an expression $e \in \mathcal{L}$ as well as a structure $M \in \mathcal{M}$. A program P navigates up and down on e using a set of pointers to move from children to parent and parent to children in order to evaluate the semantics of e over the structure M and verify that $M \models e = d$ for some $d \in D$.

A.0.1   Parameters. To write a Facet program we specify two things: (1) the language over which the program is to operate and (2) the program's auxiliary *states*, which correspond to semantic aspects, i.e. the auxiliary information used in the definition of the language semantics[8]. Part (1) involves specifying (a) the syntax trees for $\mathcal{L}$ in terms of a ranked alphabet Δ and (b) the signature for structures $\mathcal{M}$, including a set of functions used to access the data for a given $M \in \mathcal{M}$. Additionally, in part (a) we specify an algebraic data type (ADT) that endows the symbols of Δ with extra structure in order to allow programs to pattern match over the alphabet. As an example, suppose we model universal quantification in first-order logic by adding to Δ a unary symbol "$\forall x$" for each variable x from some finite set of variables. We could then treat "$\forall$" as a unary constructor to allow a program to match over all alphabet symbols that represent a universally-quantified variable. We abuse notation and write Δ for both the ranked alphabet and ADT when the context is clear.

Part (2) is accomplished by specifying an ADT for the program states. This ADT will typically be infinite, but any fixed structure will use only a finite subset of it, provided the language is FAC. We denote the state ADT by Asp and the subset pertaining to a given structure M by Asp(M). In some cases, Asp has little or no structure. For instance, in modal logic it consists of nullary constructors for the nodes of a Kripke structure. In other cases, e.g., regular expressions (Section 5), we use a *pair* constructor over positions in finite words with Asp $= \{(i, j) \in \mathbb{N} \times \mathbb{N} : i \le j\}$. We will clarify these choices in each setting as needed.

The ADTs are each associated with a set of *patterns*, which are built using the ADT constructors together with variables from a set *Var*. We use $x, z \in Var$ as pattern variables; these should not be confused with variables used in expressions $\mathcal{L}$. The sets of state and alphabet patterns are denoted by Asp(Var) and $\Delta(Var)$, respectively. Note that we do not assume Asp has a finite signature. All we will need is that Asp(M) is finite for every M and that state and alphabet pattern matching is computable. For the latter, we assume a computable function *match* that computes a unifying substitution for two members of Asp(Var) or $\Delta(Var)$ whenever possible.

A.0.2   Semantics. The semantics for Facet is given in Figure 9. It defines two relations

$$(M, n, C, \sigma) \Downarrow_p \qquad \text{and} \qquad (M, n, C, e) \Downarrow_e .$$

The predicate $\Downarrow_p$ holds for program configurations (M, n, C, σ), with C being the set of clauses for the program, M a structure, n a pointer, and $\sigma \in$ Asp(M) a state. The predicate $\Downarrow_e$ holds for expression configurations (M, n, C, e), with e being an expression from the grammar in Figure 3. The assertion $(M, n, C, \sigma) \Downarrow_p$ can be read as follows: over the structure M and syntax tree pointed to by n, the program consisting of clauses C terminates with success when started in the state σ. Proofs for $\Downarrow_p$ involve finding the matching clause for a state σ and building a subproof for $\Downarrow_e$ for the appropriate case of the `match` statement in the matching clause.

Next we discuss some details for the language.

---

[8]We sometimes use *states* and *aspects* interchangeably. But, occasionally we use *aspects* to distinguish auxiliary semantic information, with which we need not associate any operational meaning, from the operational meaning associated with *states* in the context of a program.

A.0.3   Well-Formed Programs. We consider only *well-formed* programs, which have *disjoint* and *exhaustive* clauses. That is, for every $M \in \mathcal{M}$ and $\sigma \in \mathrm{Asp}(M)$, there is precisely one clause c for which $match(pat_c, \sigma)$ succeeds, where pat_c denotes the state pattern for the clause c.

We emphasize that variables in FACET programs are *never* bound to FACET expressions. They are instead replaced either by components of the syntax tree data type or the state data type. For example, a variable x might be bound to position 2 in the word $w = abc$, while a variable y might be bound to the letter b. Variables are *bound* in FACET programs by the state pattern at the beginning of each clause, by the alphabet patterns in each case of a `match` statement, and in `all` and `any` expressions. Well-formed programs do not have free variables.

A.0.4   Computable Function Parameters. The functions $g \in S$ appearing in `any` and `all` expressions compute finite sets, e.g., elements or sets of elements from the domain of the structure. These elements are then bound to variables in `any` and `all` expressions. For example, in a totally-ordered structure like a word, we could use the function $g(l, r) = [l, r]$, or a variant $g'(l, r) = [l + 1, r]$ to compute sets of consecutive positions. For convenience we allow functions g to occur in expressions that denote states. For example, if states are pairs of word positions, we may write $(2, g(x))$, which can be evaluated to a state once x is bound. The condition in the premise of the CALL rule in Figure 9 uses a function called *norm* to reduce an expression like this to a state. The functions $g \in S$ and $f \in B$ in fact represent a family of functions, one for each $M \in \mathcal{M}$. We write $eval(M, g(v))$ and $eval(M, f(v))$ to denote the result of computing f and g in a structure M with arguments v.

A.0.5   Negation. The reader may wonder why there is no negation in the expression syntax for FACET and why the conditional for `if` expressions does not allow an arbitrary expression. These apparent restrictions are only for simplicity. If we wanted to include negation and more general conditionals, we could augment the states of any FACET program to track the parity of the number of negations seen at any point in a program execution; but we prefer to keep this complexity out of the semantics. The effect of negation can easily be accomplished by writing dual programs that implement dual operations in particular states, as described in Section 4.4.

$$\text{PROG} \quad \frac{\tau = match(c_i, \sigma) \qquad \tau' = match_{c_i}(\tau(\alpha_k), n.l) \qquad (M, n, C, \tau'(\tau(e_k))) \Downarrow_e}{(M, n, C = \{\, c_1 \dots c_m \,\}, \sigma) \Downarrow_p}$$

$$\text{CALL} \quad \frac{(M, n.c, C, \sigma') \Downarrow_p \qquad \sigma' = norm(M, \sigma(v))}{(M, n, C, \mathsf{P}(M,\ \sigma(v),\ c)) \Downarrow_e} \qquad\qquad \text{BOOL} \quad \frac{eval(M, f(v)) = \top}{(M, n, C, f(v)) \Downarrow_e}$$

$$\text{AND} \quad \frac{(M, n, C, e) \Downarrow_e \qquad (M, n, C, e') \Downarrow_e}{(M, n, C, e \text{ and } e') \Downarrow_e} \qquad\qquad \text{TRUE} \quad \frac{}{(M, n, C, \mathsf{True}) \Downarrow_e}$$

$$\text{THEN} \quad \frac{(M, n, C, e_1) \Downarrow_e \qquad eval(M, f(v)) = \top}{(M, n, C, \mathsf{if}\ f(v)\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) \Downarrow_e} \qquad \text{OR1} \quad \frac{(M, n, C, e) \Downarrow_e}{(M, n, C, e \text{ or } e') \Downarrow_e}$$

$$\text{ELSE} \quad \frac{(M, n, C, e_2) \Downarrow_e \qquad eval(M, f(v)) = \bot}{(M, n, C, \mathsf{if}\ f(v)\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) \Downarrow_e} \qquad \text{OR2} \quad \frac{(M, n, C, e') \Downarrow_e}{(M, n, C, e \text{ or } e') \Downarrow_e}$$

$$\text{ALL} \quad \frac{(M, n, C, \{x \mapsto v_1\}(e)) \Downarrow_e \quad \cdots \quad (M, n, C, \{x \mapsto v_l\}(e)) \Downarrow_e \quad eval(M, g(v)) = \{\, v_1 \ \dots \ v_l \,\}}{(M, n, C, \mathsf{all}\ (\lambda x.\ e)\ g(v)) \Downarrow_e}$$

$$\text{ANY} \quad \frac{(M, n, C, \{x \mapsto v_i\}(e)) \Downarrow_e \qquad v_i \in eval(M, g(v))}{(M, n, C, \mathsf{any}\ (\lambda x.\ e)\ g(v)) \Downarrow_e}$$

Fig. 9. Semantics for FACET. If the node $n.c$ does not exist, then the CALL rule does not apply.

A.1 Translating FACET Programs to Two-way Tree Automata

If for every structure M, the states $\mathrm{Asp}(M)$ are computable and finite, then a FACET program can be translated to a two-way alternating tree automaton over syntax trees. FACET draws attention to a small subset of tree automata that serve as semantic evaluators, and which have state spaces and alphabets that can be highly structured. Most FACET programs we have considered consist of at most a few clauses, with each clause handling the semantics for a large set of related states.

Two-way alternating tree automata over states Q and alphabet Δ assign to each state and symbol a positive Boolean formula like the following

$$\delta(q, a) \ = \ (q'', 0) \vee (q', -1) \wedge (q, 1) \wedge (q, 2).$$

This formula stipulates that in state q reading symbol a, the automaton can *either* continue from the current position (0) in state q'' or else it should succeed in several new positions: from the parent (-1) in state q' *and* from the left (1) and right (2) children, both in state q. More generally, the transitions are of the following form:

$$\delta(q, a) \in \mathcal{B}^+(Q \times \{-1, 0, ..., k\}) \qquad \text{where } q \in Q, \ a \in \Delta, \ k = \mathsf{arity}(a),$$

and they describe the viable next states and directions for the machine. It can either move up (-1), down (numbers > 0), or stay at the same node (0) on the input tree while changing state. Satisfying Boolean assignments to the set $Q \times \{-1, 0, ..., k\}$ describe strategies to build accepting automaton runs along an input syntax tree, with the two components of Q and $\{-1, 0, ..., k\}$ corresponding to the *next state* and the *direction to move*, respectively. We next explain how programs in FACET have a simple, straightforward translation to such automata.

Given a well-formed program P, a structure M, the set $\text{Asp}(M)$, and a distinguished state $\sigma_i \in \text{Asp}(M)$, we can build a two-way alternating tree automaton $\mathcal{A}(P, M)$ that accepts precisely the syntax trees (rooted at n) for which $(M, n, C(P), \sigma_i) \Downarrow_p$ holds. The states of $\mathcal{A}(P, M)$ are the members of $\text{Asp}(M)$ and the initial state is σ_i. For each clause $c \in C(P)$ of the form

$$P(M, \ \sigma(z), \ n) := \texttt{match } n.l \texttt{ with } \alpha_1 \ \rightarrow \ e_1 \ \ldots \ \alpha_m \ \rightarrow \ e_m$$

the translation creates a set of transitions. For each matching $\sigma \in \text{Asp}(M)$ there is a transition for each $a \in \Delta$. We write $\tau = match_c(\alpha_i, a)$ to mean that a matches the alphabet pattern α_i in c with unifying substitution τ and it does not match α_j for any $j < i$. We write $\tau(e)$ for the application of τ to e, which substitutes $\tau(x)$ for all free occurrences of x in e, for all x in the domain of τ.

Suppose $\tau = match(pat_c, \sigma)$ for some clause c with cases $\alpha_i \ \rightarrow \ e_i$. For each symbol $a \in \Delta$, with $\tau' = match_c(\tau(\alpha_i), a)$, we have the transition

$$\delta(\sigma, a) = \text{aut}(\tau'(\tau(e_i))),$$

with aut described below. Any $a \in \Delta$ for which no alphabet pattern matches, and which is therefore not covered above, is assigned $\delta(\sigma, a) = \bot$. Thus a $\texttt{match}$ statement need not specify what to do for alphabet symbols that, say, must never be read in specific states of the program.

Expressions e are translated into positive Boolean formulae using the function aut defined below.

$$
\begin{aligned}
\text{aut}(\texttt{True}) &= \top & \text{aut}(e_1 \texttt{ and } e_2) &= \text{aut}(e_1) \wedge \text{aut}(e_2) \\
\text{aut}(\texttt{False}) &= \bot & \text{aut}(e_1 \texttt{ or } e_2) &= \text{aut}(e_1) \vee \text{aut}(e_2) \\
\text{aut}(f(v)) &= eval(M, f(v)) & & \\
\text{aut}(\texttt{if } f(v) \texttt{ then } e_1 \texttt{ else } e_2) &=
\begin{cases}
\text{aut}(e_1) & \text{if } eval(M, f(v)) = \top \\
\text{aut}(e_2) & \text{else}
\end{cases} \\
\text{aut}(\texttt{all } (\lambda x. \ e) \ g(v)) &= \textstyle\bigwedge_{v' \in eval(M, g(v))} \text{aut}(\{x \mapsto v'\}(e)) \\
\text{aut}(\texttt{any } (\lambda x. \ e) \ g(v)) &= \textstyle\bigvee_{v' \in eval(M, g(v))} \text{aut}(\{x \mapsto v'\}(e)) \\
\text{aut}(P(M, \ \sigma(v), \ n.dir)) &= (\sigma, dir) \quad \text{where } \sigma = norm(M, \sigma(v))
\end{aligned}
$$

We can now state one other well-formedness condition for Facet programs. Namely, for every M, $\sigma \in \text{Asp}(M)$, $a \in \Delta$, and clause c such that $\tau = match(pat_c, \sigma)$, if $\tau' = match_c(\tau(\alpha_i), a)$ for the case $\alpha_i \ \rightarrow \ e_i$ in c, then we must have:

$$\text{aut}(\tau'(\tau(e_i))) \in \mathcal{B}^+(\text{Asp} \times \{-1, 0, \ldots, \text{arity}(a)\}).$$

In other words, the movement of a program along the syntax tree must respect symbol arities.

A.2 Example Construction

Here we explicitly show the construction of an automaton for a restricted regular expression language on the alphabet $\{a, b\}$. Assume regular expressions as in Section 5 but without union, intersection, and negation. Our evaluator simplifies as follows:

```
Reg(w, (l,r), n) := match n.l with
   *  → if (l = r) then True else
           any (λx. Reg(w, (l,x), n.c₁) and Reg(w, (x,r), n.stay)) [l + 1, r]
   ·  → any (λx. Reg(w, (l,x), n.c₁) and Reg(w, (x,r), n.c₂)) [l,r]
   x  → r = l + 1 and w(l) = x
```

Consider the word $w = abb$. The resulting two-way alternating automaton $\mathcal{A}(\text{Reg}, w)$ is constructed as follows. See [Comon et al. 2007] for definitions and results for such automata. The state

set is

$$Q = \{(1, 1), (1, 2), (1, 3), (1, 4), (2, 2), (2, 3), (2, 4), (3, 3), (3, 4), (4, 4),$$
$$dual(1, 1), dual(1, 2), dual(1, 3), dual(1, 4), dual(2, 2), dual(2, 3),$$
$$dual(2, 4), dual(3, 3), dual(3, 4), dual(4, 4)\}.$$

The initial state is $(1, |abb| + 1) = (1, 4)$. Here are just some of the many transitions:

$\delta((1, 2), a) = \top$
$\delta((2, 3), b) = \top$
$\delta((3, 4), b) = \top$
$\delta((1, 4), \cdot) = ((1, 1), 1) \wedge ((1, 4), 2) \vee ((1, 2), 1) \wedge ((2, 4), 2) \vee ((1, 3), 1) \wedge ((3, 4), 2)$
$\qquad \vee ((1, 4), 1) \wedge ((4, 4), 2)$
$\delta((1, 3), \cdot) = \cdots$
$\delta((1, 2), \cdot) = \cdots$
$\delta((1, 1), \cdot) = ((1, 1), 1) \wedge ((1, 1), 2)$
$\delta((2, 4), \cdot) = \cdots$
$\delta((2, 3), \cdot) = \cdots$
$\delta((2, 2), \cdot) = \cdots$
$\delta((3, 4), \cdot) = \cdots$
$\delta((3, 3), \cdot) = \cdots$
$\delta((4, 4), \cdot) = \cdots$
$\delta((1, 1), *) = \top$
$\delta((2, 2), *) = \top$
$\delta((3, 3), *) = \top$
$\delta((4, 4), *) = \top$
$\delta((1, 4), *) = ((1, 2), 1) \wedge ((2, 4), 0) \vee ((1, 3), 1) \wedge ((3, 4, 0)) \vee ((1, 4), 1) \wedge ((4, 4, 0))$
$\delta((2, 4), *) = \cdots$
$\delta((3, 4), *) = \cdots$
$\delta((1, 3), *) = \cdots$
$\delta((2, 3), *) = \cdots$
$\delta((1, 2), *) = \cdots$

All other transition formulas not already suggested above are $\bot$.

B DUAL PROGRAMS

The dual of a FACET expression e is computed as follows:

$dual(\texttt{True}) = \texttt{False}$
$dual(\texttt{False}) = \texttt{True}$
$dual(f(v)) = \neg f(v)$
$dual(P(M,\ \sigma(z),\ n.dir)) = P(M,\ flip(\sigma(z)),\ n.dir)$
$dual(e\ \texttt{and}\ e') = dual(e)\ \texttt{or}\ dual(e')$
$dual(e\ \texttt{or}\ e') = dual(e)\ \texttt{and}\ dual(e')$
$dual(\texttt{all}\ (\lambda z.\ e)\ g(v)) = \texttt{any}\ (\lambda z.\ dual(e))\ g(v)$
$dual(\texttt{any}\ (\lambda z.\ e)\ g(v)) = \texttt{all}\ (\lambda z.\ dual(e))\ g(v)$
$dual(\texttt{if}\ f(v)\ \texttt{then}\ e_1\ \texttt{else}\ e_2) = \texttt{if}\ f(v)\ \texttt{then}\ dual(e_1)\ \texttt{else}\ dual(e_2)$

where $flip(dual(\sigma(z))) = \sigma(z)$ and otherwise $flip(\sigma(z)) = dual(\sigma(z))$.

C COMPUTATION TREE LOGIC

We want a semantic evaluator for CTL syntax trees φ over pointed Kripke structures $G = (W, s, E, P)$ that checks whether $G \models \varphi$. Like modal logic, the semantic aspects again include nodes of G, but now also a counter to interpret path quantifiers recursively.

We consider the following grammar for CTL formulas, from which the other standard operators can be defined:

$$\varphi ::= a \in \Sigma \mid \varphi \vee \psi \mid \neg\varphi \mid \mathsf{EG}\varphi \mid \mathsf{E}(\varphi\mathsf{U}\psi) \mid \mathsf{EX}\varphi$$

The semantics for path quantifiers is given recursively based on the following:

$$G, w \models \mathsf{EX}\varphi \iff \exists w'.\, E(w, w') \text{ and } G, w' \models \varphi$$

with the other two path quantifiers interpreted according to the equivalences

$$(1) \quad \mathsf{EG}\varphi \equiv \varphi \wedge \mathsf{EX}(\mathsf{EG}\varphi) \quad \text{and} \quad (2) \quad \mathsf{E}(\varphi\mathsf{U}\psi) \equiv \psi \vee \varphi \wedge \mathsf{EX}(\mathsf{E}(\varphi\mathsf{U}\psi)),$$

with (1) understood as a greatest fixpoint and (2) as a least fixpoint, as we discuss shortly. The CTL program is given in Figure 10. Below, and in the program, we write "Ew" as a shorthand for $\{w' \in W \,:\, E(w, w')\}$. Fix a Kripke structure $G = (W, s, E, P)$.

The alphabet ADT is trivial. The state ADT consists of two parts:

$$\mathsf{Asp}(G) := \{w, dual(w) \,:\, w \in W\} \sqcup \mathsf{Count}(G)$$
$$\mathsf{Count}(G) := \{(w, i), (dual(w), i) \,:\, w \in W, 0 \leq i \leq |W|\},$$

the first being states of the form w or $dual(w)$, which do not involve a counter, and the second being states of the form (w, i) or $(dual(w), i)$, which use a counter to verify path quantifiers.

The counter value i tracks the stages of a least or greatest fixpoint computation. If the counter is being used to verify a formula $\mathsf{EG}\varphi$ holds at some $w \in W$, then this is a greatest fixpoint. On the other hand, if the counter is being used to verify a formula $\mathsf{E}(\varphi\mathsf{U}\psi)$ does *not* hold at some $w \in W$, then it is a least fixpoint. To understand this, let us consider the equivalences (1) and (2) as monotone functions over $\mathcal{P}(W)$. We write $[\![\varphi]\!]_G := \{w \in W \,:\, G, w \models \varphi\}$ for the set of states where a given formula holds. Now observe that (1) and (2) correspond to the monotone functions

$$\mathsf{EG}_\varphi(X) \quad = \quad [\![\varphi]\!] \quad \cap \quad \{w \in W \,:\, Ew \cap X \neq \emptyset\}$$
$$\mathsf{EU}_{\varphi,\psi}(X) \quad = \quad [\![\psi]\!] \quad \cup \quad \{w \in W \,:\, Ew \cap X \neq \emptyset\} \cap [\![\varphi]\!],$$

with $[\![\mathsf{EG}\varphi]\!]_G = \mathsf{gfp}(\mathsf{EG}_\varphi)$ and $[\![\mathsf{E}(\varphi\mathsf{U}\psi)]\!]_G = \mathsf{lfp}(\mathsf{EU}_{\varphi,\psi})$. The program CTL in Figure 10 has the property that for all $G = (W, s, E, P)$, $w \in W$, $0 \leq i \leq |W|$, and φ:

$$(G, root(\mathsf{EG}\varphi), C(\mathsf{CTL}), (w, i)) \Downarrow_p \iff w \in \mathsf{EG}_\varphi^{|W|-i}(W)$$
$$\text{and} \quad (G, root(\mathsf{E}(\varphi\mathsf{U}\psi)), C(\mathsf{CTL}), (dual(w), i)) \Downarrow_p \iff w \notin \mathsf{EU}_{\varphi,\psi}^{|W|-i}(\emptyset).$$

The task of evaluating $w \notin [\![\mathsf{EG}\varphi]\!]_G$ needs no counter because there is always a finite computation witnessing non-membership for EG-formulas. The task of evaluating whether $w \in [\![\mathsf{E}(\varphi\mathsf{U}\psi)]\!]_G$ needs no counter for the same reason. There are always proofs of *removal* from a greatest fixpoint and proofs of *inclusion* in a least fixpoint. Note we do not mind infinite loops in the other cases because we interpret programs as tree automata with reachability acceptance.

THEOREM C.1. *CTL separation for finite sets P and N of finite pointed Kripke structures, and grammar $\mathcal{G}$, is decidable in time $\mathcal{O}(2^{poly(mn^2)} \cdot |\mathcal{G}|)$, where $m = |P| + |N|$ and $n = \max_{G \in P \cup N} |G|$.*

PROOF. We have $|\mathsf{Asp}(G)| = \mathcal{O}(|W|^2)$, and the rest follows by Theorem 4.3 and Corollary 4.4. $\square$

```
CTL(G, w, n) :=
  match n.l with
    EG → CTL(G, w, 0, n.stay)
    EU → CTL(G, w, n.c₂) or
          (CTL(G, w, n.c₁) and (any (λz. CTL(G, z, n.stay)) Ew))
    EX → any (λz. CTL(G, z, n.c₁)) Ew
    ∨ → CTL(G, w, n.c₁) or CTL(G, w, n.c₂)
    ¬ → CTL(G, dual(w), n.c₁)
    x → x ∈ P(w)

CTL(G, dual(w), n) :=
  match n.l with
    EU → CTL(G, dual(w), 0, n.stay)
    EG → CTL(G, dual(w), n.c₁) or (all (λz. CTL(G, dual(z), n.stay)) Ew)
    EX → all (λz. CTL(G, dual(z), n.c₁)) Ew
    ∨ → CTL(G, dual(w), n.c₁) and CTL(G, dual(w), n.c₂)
    ¬ → CTL(G, w, n.c₁)
    x → x ∉ P(w)

CTL(G, w, i, n) :=
  match n.l with
    EG → if i = |W| then True
          else CTL(G, w, n.c₁) and (any (λz. CTL(G, z, i+1, n.stay)) Ew)

CTL(G, dual(w), i, n) :=
  match n.l with
    EU → if i = |W| then true
          else CTL(G, dual(w), n.c₂) and
                (CTL(G, dual(w), n.c₁) or
                 (all (λz. CTL(G, dual(z), i+1, n.stay)) Ew))
```

Fig. 10. CTL evaluates CTL formulas φ against an input pointed Kripke structure G and checks $G \models \varphi$.

D FINITE-VARIABLE FIRST-ORDER LOGIC

Fix a finite relational signature with relation symbols R_i and a set of variables $V = \{x_1, \ldots, x_k\}$. We write a program FO that evaluates an FO^k syntax tree φ against a relational structure M and checks $M \models \varphi$. The aspects are the (partial) assignments to variables V:

$$\text{Asp}(M) := \{\gamma, dual(\gamma) \ : \ \gamma \in [V \rightharpoonup M]\},$$

and $|\text{Asp}(M)| = \mathcal{O}(|M|^k)$. The alphabet ADT consists of unary constructors for $\forall$ and $\exists$, as well as l-ary constructors for each l-ary relation symbol R. The program FO together with its (omitted) dual allows us to derive the main result of [Krogmeier and Madhusudan 2022]. The other results can also be derived, e.g., FO^k with recursive definitions can be interpreted using a combination of two-way navigation as in Section 7 and counters as in Appendix C.

```
FO(M, γ, n) :=
  match n.l with
    ∧    → FO(G, γ, n.c₁) and FO(G, γ, n.c₂)
    ∨    → FO(G, γ, n.c₁) or FO(G, γ, n.c₂)
    ¬    → FO(G, dual(γ), n.c₁)
    ∀x   → all (λz. FO(G, z, n.c₁)) {γ[x ↦ a] : a ∈ M}
    ∃x   → any (λz. FO(G, z, n.c₁)) {γ[x ↦ a] : a ∈ M}
    R(x̄) → γ(x̄) ∈ Rᴹ
```

Fig. 11. FO evaluates first-order logic formulas φ against an input relational structure M and checks $M \models \varphi$.