Learning and Logic for Formal Synthesis

Benjamin Caulfield

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2021-113
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-113.html

May 14, 2021

Acknowledgement

This thesis is dedicated to my parents. Without their continued support for my education it would never have been written.

I would like to sincerely thank my advisor, Sanjit Seshia, who has guided me throughout my graduate career. Thanks as well to the other members of my thesis committee Tom Griffiths and Alessandro Chiesa for their time and support.

Finally, thank you to the members of my qualifying exams Stavros Tripakis and Loris D'Antoni, who could not be on my committee as they work at other universities. During my first years at Berkeley, Stavros also served as my co-advisor. My first projects were with him and I thank him for helping me gain the initial confidence I needed to complete this degree.

Learning and Logic for Formal Synthesis

by

Benjamin Caulfield

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Sanjit A. Seshia, Chair
Alessandro Chiesa
Thomas L. Griffiths

Spring 2021

The dissertation of Benjamin Caulfield, titled Learning and Logic for Formal Synthesis, is approved:

Chair _____ Date _____

_____ Date _____

_____ Date _____

University of California, Berkeley

Abstract

Learning and Logic for Formal Synthesis

by

Benjamin Caulfield

Doctor of Philosophy in Computer Science

University of California, Berkeley

Sanjit A. Seshia, Chair

Program synthesis is the use of algorithms to derive programs that satisfy given specifications. These specifications are usually given in some form of computer-understandable logic. Specifications are usually much easier to write than the programs themselves.

By 'filling in the details' given by the specification, program synthesis opens the possibility of creating simple programs to both laypeople and non-programming domain experts.

Recent work in program synthesis has used techniques from 'exact active learning', where learning algorithms are able to pose queries to oracles. In the case of program synthesis, these oracles are implemented by checking potential programs against the given specification or asking a user for more inputs.

Another recent development in the field is Syntax-Guided Synthesis (SyGuS), where the space of potential programs is given by a tree-grammar (or context-free grammar). Specifications are given in the logic of SMT (satisfiability modulo theories) problems.

This thesis further develops the theory behind exact active learning, program synthesis, and their intersection. We provide upper and lower bounds, including undecidability results, for SyGuS problems defined on various SMT theories. We introduce the subject of actively learning equational theories and show how it can be used to learn constrained EUF formulas. We study the problem of exact active learning of concepts that are comprised of independent components, and show when the knowledge that components work independently can significantly reduce learning complexity. Finally, we introduce new methods for SyGuS solving with respect to cost, where the goal is to find the minimal cost program satisfying a specification.

To my parents

Contents

List of Figures

List of Tables

Acknowledgments

This thesis is dedicated to my parents. Without their continued support for my education it would never have been written.

I would like to thank my advisor, Sanjit Seshia, who has guided me throughout my graduate career. Thanks as well to the other members of my thesis committee Tom Griffiths and Alessandro Chiesa for their time and support.

Finally, thank you to the members of my qualifying exams Stavros Tripakis and Loris D'Antoni, who could not be on my committee as they work at other universities. During my first years at Berkeley, Stavros also served as my co-advisor. My first projects were with him and I thank him for helping me gain the initial confidence I needed to complete this degree.

Chapter 1

Introduction

1.1 Program Synthesis

The key idea behind program synthesis is that specifications are easier to write than programs. Users can write program specifications in high-level specification languages which can be understood by computers. For example, consider the following specification for the square root function $sqrt$ adapted from [42]:

$$sqrt(n) = m \text{ where } (m \geq 0) \text{ and } (m^2 = n)$$

This specification is easy enough to understand: the square root of n is a non-negative number m whose square is n. The algorithm to actually calculate square root is of course much more difficult than this one-line specification. A successful synthesis algorithm could take this specification and find the square root algorithm. This opens the possibility of creating simple programs to both laypeople and non-programming domain experts.

One important technique for program synthesis OGIS, which will be defined below, can also be used to make programs clearer and more efficient. Given an unclear or inefficient program, a program synthesis algorithm can make black-box queries to the program in order to learn a clearer or more efficient program [33]. This unclear program might even take the form of a neural network. Extending work on learning automata from neural networks to learning general programs, could mean that program synthesis might play an important role in the explanation and verification of neural network behaviors.

Past Work

The first successful instance of program synthesis was demonstrated by Green in 1969, who showed that theorem proving techniques could be used to solve simple problems like the Tower of Hanoi [29]. Later, Waldinger & Manna combined techniques from unification, rewriting, and mathematical induction to create a synthesis framework that stood as the state-of-the-art for decades [42].

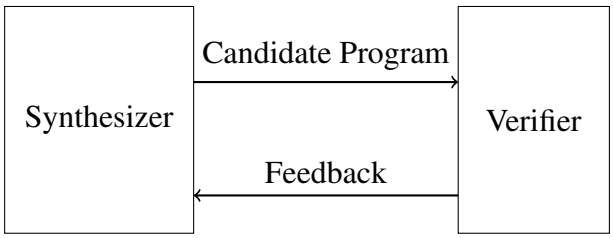

Figure 1.1: Inductive synthesis procedure

In addition to the logical specifications given above, program synthesis can also require that a program match a set of examples representing the way a program should (or should not) work. This type of synthesis is called *inductive synthesis*, whereas the former type is called *deductive synthesis*.

In 2005, Solar-Lezama, Rabbah, Bodík, and Ebcioglu introduced *sketching*, where a partial program specification is written with holes, which are then filled in by a synthesizer [54]. This was expanded to general combinatorial sketching in 2006 by Solar-Lezama Tancau, Bodik, Seshia, and Saraswat [53]. This was later developed by Solar-Lezama into the general synthesis procedure called counterexample-guided inductive synthesis (CEGIS) as in Figure 1.1 [51]. In this paradigm, potential programs are compared against the program specification using a verifier. If the program is incorrect, the verifier returns a counterexample, meaning an input that causes incorrect program behavior. This counterexample is then used to find a new potential program, and the process continues.

The exact active learning paradigm learns by posing queries to an oracle that can answer queries about a target concept, which is drawn from a class of possible concepts. The focus for this paradigm is to find a solution that *exactly* matches the target concept. This is particularly useful for the safety-critical applications often studied in formal methods. A well-known exact active learning algorithm is Angluin's polynomial time algorithm for learning finite automata from queries and counterexamples [5].

In the case of CEGIS, the oracle was answering equivalence queries, by giving counterexamples showing that the hypothesis concept did not fit the intended solution. By implementing other oracles, new classes of programs could be quickly learned. In 2012, Seshia showed that active learning algorithms can be integrated with deductive methods to create effective synthesis procedures [49, 48]. Another early work on oracle-guided synthesis, written by Jha, Gulwani, Seshia, and Tiwari, showed that a programmer could write an inefficient version of a program, which a synthesis algorithm could then query in order to learn a better version [34]. A program synthesis algorithm could also run a difficult-to-read program on different inputs in order to learn a simpler representation of that program. The use of oracles for program synthesis was further studied in Jha's PhD thesis [35] and later formalized by Jha & Seshia as oracle guided inductive synthesis (OGIS), which is the main form of program synthesis studied in this paper [33]. OGIS is a special case of *formal synthesis*, the synthesis of programs that are correct with respect to a high-level specification.

Gulwani, Jha, Tiwari, and Venkatesan have shown that non-trivial programs can be built from

loop-free compositions of simpler components [30]. In 2013, Alur et. al. introduced Syntax-Guided Synthesis (SyGuS) as a framework to unify existing synthesis techniques, including component-based synthesis and sketching [4]. SyGuS is a general framework for synthesizing loop-free programs that match given specifications which are given in SMT logics. The space of potential programs is given by a context-free grammar (or tree grammar). By cleverly searching this space, SyGuS solvers can find terms (i.e., loop-free programs) that match the given specification.

1.2 Contributions

This work presents several new results at the intersection of program synthesis and exact active learning, including new algorithms and theoretical bounds.

New Algorithms

We present several new algorithms, often for newly studied problems. These include new learning algorithms for equational theories, as in Chapter 3, and modular learning algorithms as in Chapter 4. We also provide terminating algorithms for several classes of SyGuS problems in Chapter 5. Among these problems are finite domain theories and a newly introduced subset of EUF called *regular-EUF*. Finally, we show new ways to solve SyGuS problems with respect to cost. Chapter 6 argues how an enumeration algorithm from the natural language processing community can be used to find min-cost SyGuS solutions with respect to certain classes of costs. It also shows how SyGuS grammars can be adapted to acount for several types of useful costs, including the big-O runtime and space usage of programs.

Theoretical Bounds

This thesis also presents theoretical bounds, showing limits to the possibilities of learning algorithms and synthesizers. Chapter 4 presents several bounds on the limitations of modular learning of crossproducts with respect to different sets of queries. The undecidability of SyGuS with respect to EUF, Arrays, and unbounded Bitvectors is shown in Chapter 5.

1.3 Organization

The next chapter introduces background material. After this, the thesis is organized into two parts. Part 1 focuses on results in computational learning theory. Chapter 3 introduces results on learning equational theories and is based on work with Ashish Tiwari. Chapter 4 studies the problem of modular concept learning and was developed with Sanjit A. Seshia.

Part 2 discuses new results in Syntax-Guided Synthesis. Chapter 5 gives decidability results for SyGuS with respect to different background theories and was work with Sanjit A. Seshia, Stavros Tripakis, and Markus Rabe. Chapter 6 discusses methods for expanding SyGuS to account for costs and was developed with Nick Spooner, Morris Yau, and Sanjit A. Seshia.

Chapter 2

Background

2.1 Notation

Symbol	Meaning
Σ	Alphabet (ranked or unranked)
$T(\Sigma, V)$	Set of terms that can be made with symbols Σ and variables V
$T(\Sigma)$	The set $T(\Sigma, \emptyset)$
G	Tree Grammar or String Grammar
$L(G), L(A)$	The set of trees or strings produced by grammar G or automaton A
ϕ, ψ, Φ, Ψ	Formulas
X, Y	Space of elements / Instance space
2^S	Power set: set of all subsets of S
C	Concept class ($C \subseteq 2^X$)
c	Concept ($c \in C$ and $c \subseteq X$)
c^*	Target concept. The concept to be learned ($c^* \in C$)
$\mathcal{D}$	Probability distribution defined on X
M	First-order model
$dom(M)$	Domain of model M
$(-)^M$	Interpretation of M

Table 2.1: Mathematical Notation

Symbol	Meaning
DFA	Deterministic Finite Automaton
RTG	Regular Tree Grammar
CFG	Context-Free Grammar
PAC	Proably Approximately Correct

Table 2.2: Abbreviations

The above two tables give notation and abbreviations that are used throughout the thesis. Table 2.1 defines mathematical notation and shows conventions for variable names. Table 2.2 shows acronyms and abbreviations.

The remainder of this section reviews some key definitions and results used in the rest of the thesis.

Terms and Substitutions

We follow the book by Baader and Nipkow [8]. A *signature* (or *ranked alphabet*) Σ consists of a set of *function* symbols with an associated *arity*, a non-negative number indicating the number of arguments. For example, $\Sigma = \{f : 2, a : 0, b : 0\}$ consists of binary function symbol f and constants a and b. For any arity $n \geq 0$, we let $\Sigma^{(n)}$ denote the set of function symbols with arity n (the n-ary symbols). We will refer to the 0-ary function symbols as *constants*.

For any signature Σ and set of *variables* V such that $\Sigma \cap V = \emptyset$, we define the set $T(\Sigma, V)$ of Σ-terms over V inductively as the smallest set satisfying:

- $\Sigma^{(0)}, V \subseteq T(\Sigma, V)$
- For all $n \geq 1$, all $f \in \Sigma^{(n)}$, and all $t_1, \ldots, t_n \in T(\Sigma, X)$, we have $f(t_1, \ldots, t_n) \in T(\Sigma, V)$.

We define the set of *ground terms* of Σ to be the set $T(\Sigma, \emptyset)$ (simplified to $T(\Sigma)$). We define the subterms of a term recursively as $Subterms(g(s_1, \ldots, s_k)) = \{g(s_1, \ldots, s_k)\} \cup \bigcup_i Subterms(s_i)$, which we lift to sets S of terms, $Subterms(S) = \bigcup_{s \in S} Subterms(s)$. We say that a set S of terms is *subterm-closed* if $Subterms(S) = S$.

For a set $y_1, \ldots, y_k$ of variables (or constants) and terms $t_1, \ldots, t_k, s$, the term $s\{t_1/y_1, \ldots, t_k/y_k\}$ is formed by replacing each instance of each y_i in s with t_i. We call $\sigma := \{t_1/y_1, \ldots, t_k/y_k\}$ a *substitution*. Substitutions extend in the natural way to formulae, by applying the substitution to each term in the formula.

We extend substitution to function symbols with arity > 0, where it is also called *second-order* substitution. For a function symbol f of arity k, a signature Σ, and a set of variables $\{x_1, \ldots, x_k\}$, a *substitution* to f in Σ is a term $w \in T(\Sigma, \{x_1, \ldots, x_k\})$. Given a term $s \in T(\Sigma \cup f)$, the term $s\{w/f\}$ is formed by replacing each occurrence of any term $f(s_1, \ldots, s_k)$ in s with $w\{s_1/x_1, \ldots, s_k/x_k\}$ (sometimes written $w(s_1, \ldots, s_k)$). For example, $g(f(a, b))\{h(x_2, x_1, c)/f\}$ is equivalent to $g(h(b, a, c))$. We say that $x_1, \ldots, x_k$ are the *bound variables* of f. Intuitively,

second-order substitution replaces not only f by w, but also replaces the arguments $s_1, \ldots, s_k$ of each function application $f(s_1, \ldots, s_k)$ by the bound variables.

A *context* B is a term in $T(\Sigma, \{x\})$ with a single occurrence of x. For $s \in T(\Sigma)$, we write $B[s]$ for $B\{s/x\}$.

2.2 Logical Theories

A first-order model M in Σ, also called Σ-model, is a pair consisting of a set $\mathit{dom}(M)$ called its domain and a mapping $(-)^M$. The mapping, called the *interpretation*, assigns to each function symbol $f \in \Sigma^F$ with arity $n \geq 0$ a total function $f^M : \mathit{dom}(M)^n \to \mathit{dom}(M)$, and to each relation $R \in \Sigma^R$ of arity n a set $R^M \subseteq \mathit{dom}(M)^n$.

A *formula* is a boolean combination of relations over terms. The mapping induced by a model M defines a natural mapping of formulas $\varphi \in L(\Sigma)$ to truth values, written $M \models \varphi$ (we also say M *satisfies* φ). For some set Φ of first-order formulas, we say $M \models \Phi$ if $M \models \varphi$ for each $\varphi \in \Phi$. A *theory* $\mathcal{T} \subseteq L(\Sigma^F \cup \Sigma^R)$ is a set of formulas. We say M is a model of $\mathcal{T}$ if $M \models \mathcal{T}$, and use Mod($\mathcal{T}$) to denote the set of models of $\mathcal{T}$. A first-order formula φ is *valid* in $\mathcal{T}$ if for all $M \in \text{Mod}(\mathcal{T})$, $M \models \varphi$. A theory is *complete* if for all formulas $\varphi \in L(\Sigma)$ either φ or $\neg\varphi$ is valid.

Given a set of ground equations $E \subseteq T(\Sigma) \times T(\Sigma)$ and terms $s, t \in T(\Sigma)$, we say that $s \to_E t$ if there exists an (l, r) in E and a context C such that $C[l] = s$ and $C[r] = t$. For example, if $E := \{a = g(b)\}$, then $h(a) \to_E h(g(b))$. Let $=_E$ be the symmetric and transitive closure of $\to_E$. We will sometimes write $E \vdash s = t$ instead of $s =_E t$. We will use $[s]_E$ to represent the set $\{t \mid s =_E t\}$. ***Birkhoff's Theorem*** states that for any ranked alphabet Σ, set $E \subseteq T(\Sigma) \times T(\Sigma)$, and $s, t \in T(\Sigma)$, $E \vdash s = t$ if and only if for every model M in Σ such that $M \models \bigwedge_{e \in E} e$ it holds $M \models s = t$ [8].

In this work, we consider the common quantifier-free background theories of SMT solving: propositional logic (SAT), bit-vectors (BV), difference logic (DL), linear real arithmetic (LRA), linear integer (Presburger) arithmetic (LIA), the theory of arrays (AR), and the theory of uninterpreted functions with equality (EUF). For detailed definitions of these theories, see [10, 9].

For the theory of EUF it is common to introduce the If-Then-Else operator (*ITE*) as syntactic sugar [13, 10, 9]. We follow this tradition and allow EUF formulas to contain terms of the form $\mathit{ITE}(\varphi, t_1, t_2)$, where φ is a formula, and t_1 and t_2 are terms. To desugar EUF formulas we introduce an additional constant c_{ite} and add two constraints $\varphi \to (c_{\text{ite}} = t_1)$ and $\neg\varphi \to (c_{\text{ite}} = t_2)$ for each *ITE* term $\mathit{ITE}(\varphi, t_1, t_2)$. As we will see in Section 5.2 the presence of syntactic sugar such as the *ITE* operator in the grammar of SyGuS problems has a surprising effect on the decidability of the SyGuS problem.

2.3 Grammars and Automata

A *context-free grammar* (CFG) is a tuple $G = (N, T, S, R)$ consisting of a finite set N of non-terminal symbols with a distinguished *start symbol* $S \in N$, a finite set T of terminal symbols, and

a finite set R of production rules, which are tuples of the form $(N, (N \cup T)^*)$. Production rules indicate the allowed replacements of non-terminals by sequences over non-terminals and terminals. The *language*, $L(G)$, generated by a context-free grammar is the set of all sequences that contain only terminal symbols that can be derived from the start symbol using the production rules.

A *regular tree grammar* $G = (N, S, \Sigma, R)$ consists of a set N of non-terminals, a start symbol $S \in N$, a ranked alphabet Σ, and a set R of production rules. Production rules are of the form $A \to g(t_1, t_2, ..., t_k)$, where $A \in N$, g is in Σ and has arity k, and each t_i is in $N \cup T_\Sigma$. For a given tree-grammar G we write $L(G)$ for the set of trees produced by G. Note that we will refer to trees and terms interchangeably in this thesis. For example, $g(a, b, c)$ can be thought of as the tree with root g and children a, b, and c. The *regular tree languages* are the languages produced by some regular tree grammar. Any regular tree grammar can be converted to a CFG by simply treating the right-hand side of any production as a string, rather than a tree. Thus, the *undecidability results for SyGuS given regular tree grammars extend to undecidability results for SyGuS given CFGs*.

We believe that regular tree grammars better represent the space of possible SyGuS solutions than CFGs. Any CFG whose non-terminals can only produce well-formed strings can be easily represented as a regular tree grammar. The alternative is to have non-terminals yield strings that are somewhat unintuitive, such as in the following CFG: $S \to AB$
$A \to g(a, \quad B \to b)$.

Let Σ be a signature of a background theory $\mathcal{T}$. We define a tree grammar $G = (N, S, \Sigma, P)$ to be *$\mathcal{T}$-compatible* (or Σ-compatible) if $\Sigma \subseteq \Sigma^F \cup \Sigma^R$ and the arities for all symbols in Σ match those in Σ.

A (deterministic) bottom-up (or *rational*) tree automaton A is a tuple (Q, Σ, δ, Q_F). Here, Q is a set of states, $Q_F \subseteq Q$, and Σ is a ranked alphabet. The function δ maps a symbol $g \in \Sigma^{(k)}$ and states $q_1, \ldots, q_k$ to a new state q', for all k. If no such q' exists, $\delta(g, q_1, \ldots, q_k)$ is undefined. We can inductively extend δ to a function δ^* on terms, where for all $g \in \Sigma^{(k)}$ and all $s_1, \ldots, s_k \in T(\Sigma)$, we set $\delta^*(g(s_1, \ldots, s_k)) := \delta(g, \delta^*(s_1), \ldots, \delta^*(s_k))$. The language accepted by A is the set $L(A) := \{s \mid s \in T(\Sigma), \delta^*(s) \in Q_F\}$. There exist fast transformations between regular tree grammars and rational tree automata [18], and we will sometimes also define SyGuS problems in terms of rational tree automata rather than regular tree grammars.

2.4 Syntax-Guided Synthesis

We follow the definition of SyGuS given by Alur et al. [4]. Let $\mathcal{T}$ be a background theory over signature Σ, and let $\mathcal{G}$ be a class of grammars. Given a function symbol f with arity k, a formula φ over the signature $\Sigma \dot\cup \{f\}$, and a grammar $G \in \mathcal{G}$ of terms in $T(\Sigma, \{x_1, \ldots, x_k\})$, the SyGuS problem is to find a term $w \in L(G)$ such that the formula $\varphi\{w/f\}$ is valid or to determine the absence of such a term. We represent the SyGuS problem as the tuple $(\varphi, \mathcal{T}, G, f)$.

The variables $x_1, \ldots, x_k$ that may occur in the generated term w stand for the k arguments of f. For each function application of f the second-order substitution $\varphi\{w/f\}$ then replaces $x_1, \ldots, x_k$ by the arguments of the function application.

Note that the original definition of SyGuS allows for universally quantified variables, while our definition above admits no variables. This is equivalent, since universally quantified variables can be replaced with fresh constants without affecting validity.

Example 1. *Consider the following example SyGuS problem in linear integer arithmetic. Let the type of the function to synthesize f be int $\times$ int $\mapsto$ int and let the specification be given by the logical formula*

$$\varphi_1 : \ \forall x, y \ f(x, y) = f(y, x) \ \wedge \ f(x, y) \geq x \, .$$

We can restrict $f(x, y)$ to be expressions generated by the grammar below:

$$\begin{aligned} \mathit{Term} &:= x \mid y \mid \mathit{Const} \mid \ \mathit{ITE}(\mathit{Cond}, \mathit{Term}, \mathit{Term}) \\ \mathit{Cond} &:= \mathit{Term} \leq \mathit{Term} \mid \mathit{Cond} \wedge \mathit{Cond} \mid \neg \mathit{Cond} \mid (\mathit{Cond}) \end{aligned}$$

A function computing the maximum over x and y, such as $\mathit{ITE}(x \leq y, y, x)$, is a solution to the SyGuS problem.

2.5 Computational Learning Models

Much of this thesis focuses on *concept learning*: learning representations of sets of elements from a class of possible representations. This differs from other types of learning in that it does not allow for more than two labels and it does not attribute probabilities to those labels. An element is either in the concept or not.

An *instance space* (sometimes called *element space* or just *space*) is a set X, which may be finite or infinite. A concept in X is some set $c \subseteq X$. A concept class C over a space X is a finite or infinite collection of concepts in X. When learning concepts, we assume there is a *target concept* c^*, which it is the learning algorithm's goal to find [37].

For example, when learning regular languages, the input space X would be the set Σ^*: the set of all finite strings that can be formed from symbols in the alphabet Σ. The concept class C would be the set of all regular languages, which might be represented, for example, as deterministic finite automata ($DFAs$). An individual concept c would then be a specific regular language (i.e., set of strings) represented by a finite automaton.

Note that although the choice of representation can sometimes be crucial for concept learning, that is not the case in this thesis. We will specify a fixed representation for each concept class. For example, unless otherwise stated, regular languages will always be represented by the canonical minimal DFA.

A learning algorithm (learner) is defined over a fixed space X and concept class C. For any target concept $c^* \in C$, the algorithm has access to either queries or examples from c^* and must eventually return a representation of c^*. The complexity of learning c^* is a function of the size of the representation of c^*. Complexity measures of interest include the amount of time and space needed to learn, as well as the number of queries or examples that are needed.

Query Name	Symbol	Complexity	Oracle Definition
Single Positive Query	$1Pos$	$n\backslash a$	Return a fixed $x \in c^*$
Positive Query	Pos	$\#Pos(c^*)$	Return an $x \in c^*$ that has not yet been given as a positive example (if one exists)
Membership Query	*Mem*	$\#Mem(c^*)$	Given element x, return 'true' iff $x \in c^*$
Equivalence Query	*EQ*	$\#EQ(c^*)$	Given $c \in C$, return 'true' if $c = c^*$ otherwise return $x \in (c\backslash c^*) \cup (c^*\backslash c)$
Subset Query	*Sub*	$\#Sub(c^*)$	Given $c \in C$, return 'true' if $c \subseteq c^*$ otherwise return some $x \in c\backslash c^*$
Superset Query	*Sup*	$\#Sup(c^*)$	Given $c \in C$, return 'true' if $c \supseteq c^*$ otherwise return some $x \in c^*\backslash c$
Example Query	$EX_{\mathcal{D}}$	$\#EX(c^*, \mathcal{D})$	Samples x from distribution $\mathcal{D}$ and returns x with a label indicating whether $x \in c^*$.

Table 2.3: Types of queries studied in this thesis.

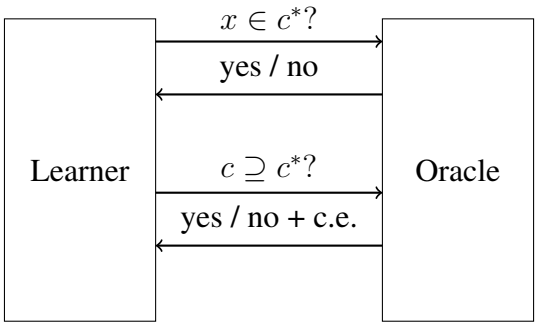

Figure 2.1: Learner and Oracle asking and answering membership and superset queries.

Query-Based Learning

Most of the learning done in this thesis will involve the use of *queries* and *oracles* as shown in Figure 2.5 [6]. Oracles are objects with knowledge of the target concept c^*. They can answer specific *queries* posed by the learning algorithm about c^*. For example, a *membership oracle* would be given an element $x \in X$ (called a *membership query*) from the learning algorithm, and would then tell the algorithm whether or not $x \in c^*$. An *equivalence query* from a learning algorithm is a set $c \in C$. The oracle would either tell the learner that c equals c^* or would give a *counterexample*, which is an element of X that demonstrates that c and c^* are not equivalent. In the case of equivalence queries, a counterexample is either a positive counterexample $x \in c^*\backslash c$ or a negative counterexample $x \in c\backslash c^*$. The set of queries discussed in this thesis are given in Table 2.3.

Part I

Learning

Chapter 3

Actively Learning Equational Theories

3.1 Overview

Term rewriting systems are among the many formalisms capable of describing all turing-computable functions. Lambda calculus, combinatory logic, and the theory of turing machines all have very natural representations as term rewriting systems. Often, the order on rewrite rules is ignored, and rewrite rules are treated as sets of unordered equations (identities) between terms. Algorithms such as the Knuth-Bendix algorithm [38] are then applied to such sets of equations to create term rewriting systems with desirable properties. Because of the close connection between these equations and programs [59, 44], applying the Knuth-Bendix algorithm to equations can be seen as a form of program-synthesis [24]. Dershowitz and Reddy have also studied the inductive synthesis of programs using ordered-rewriting [25]. Although there has also been work on learning term-rewriting systems [7, 47], there has been much less work on the learning of equational systems, themselves.

The goal of the present chapter is to learn finite sets of equations over terms using examples or queries. We focus on exact learning, where the learned equational theory must be equivalent to the target equational theory. Although we learn equations with variables, we restrict ourselves to using examples and queries on only ground equations (i.e., equations without variables). We can think of these ground equations as instantiations of non-ground equations in the "real world". This method, however, has the downside that we cannot distinguish between two different equational theories that agree on ground terms (i.e., terms without variables). For example, given alphabet $\Sigma := \{f : 1, a : 0\}$ consider the equational theories generated by $E_0 := \{f(f(a)) \approx f(a)\}$ and $E_1 := \{f(x) \approx f(y)\}$ for variables x and y. Both presentations agree on all ground equations, but the equation $f(x) \approx f(y)$ is provable in E_1 but not E_0. Therefore we only require that the learned equational theory be equal to the target theory on all ground terms.

Note that this type of learning fits into the concept learning paradigm. The instance space X is the set $T(\Sigma) \times T(\Sigma)$ of pairs between ground terms. For a given presentation, the set of such pairs that are provably equivalent forms a concept. A concept class can be formed by the set of all concepts induced by presentations, assuming the presentations follow desired restrictions. So, for

example, one concept class might be the set of concepts induced by a finite set of ground equations. Another class might be the set of concepts induced by any finite presentations. Analogous definitions of positive & negative examples and membership & equivalence queries can be applied to learning equational theories. So, in the context of learning equational theories, a membership query would mean "is this ground equation provable by the target equation".

This chapter presents polynomial-time algorithms for the exact learning of *non-collapsing shallow theories*, which are presentable by equations with variables appearing only at depth 1 . Shallow theories, which are presentable by equations with variables appearing at depth 0 or 1, were first introduced by Comon, Haberstrau, and Jouannaud as a non-trivial class of equational theories with a decidable word problem [19, 20]. Later work by Niewenhuis showed that the word problem for shallows theories is solvable polynomial time [43]. It seems unlikely that there could exist a polynomial-time learning algorithm for a class of theories with no polynomial algorithm for the word problem. Therefore, the set of non-collapsing shallow theories is among the most expressive classes of theories for which an efficient learning algorithm might reasonably exist.

3.2 Notation and Background

Terms and Substitutions

This section introduces more background on terms and equations. Refer to the background section at the start of this work for more information.

We follow the book by Baader and Nipkow [8]. A *signature* (or *ranked alphabet*) Σ consists of a set of *function* symbols with an associated *arity*, a non-negative number indicating the number of arguments. For example $\Sigma := \{f : 2, a : 0, b : 0\}$ consists of binary function symbol f and constants a and b. For any arity $n \geq 0$, we let $\Sigma^{(n)}$ denote the set of function symbols with arity n (the n-ary symbols). We will refer to the 0-ary function symbols as *constants*.

For any signature Σ and set of *variables* V such that $\Sigma \cap V = \emptyset$, we define the set $T(\Sigma, V)$ of Σ-terms over V inductively as the smallest set satisfying:

- $\Sigma^{(0)}, V \subseteq T(\Sigma, V)$
- For all $n \geq 1$, all $f \in \Sigma^{(n)}$, and all $t_1, \ldots, t_n \in T(\Sigma, V)$, we have $f(t_1, \ldots, t_n) \in T(\Sigma, V)$.

Unless otherwise stated, we will use variations of a, b, and c to denote constants, x, y, and z to denote variables, and f, g, and h to denote non-constant function symbols. We define the set of *ground terms* of Σ to be the set $T(\Sigma, \emptyset)$, which we will sometimes write $T(\Sigma)$. So ground terms are terms with no variables. Ground equations are equations between ground terms.

The set of *positions* of a term t, denoted $Pos(t)$, is a set of strings over the alphabet of positive integers. It is inductively defined as follows:

- If $t \in V \cup \Sigma^{(0)}$, then $Pos(t) := \{\epsilon\}$
- If $t = f(t_1, \ldots, t_n)$, then $Pos(t) := \{\epsilon\} \cup \bigcup_{i=1}^{n} \{ip \mid p \in Pos(t_i)\}$

Here, ϵ represents the empty string and is called the *root position* of t. For positions p and q, we say $p \leq q$ if there exists a position p' such that $pp' = q$. We say p is parallel to q, denoted $p \| q$, if $p \not\leq q$ and $q \not\leq p$. For $p \in Pos(t)$, the subterm of t at position p, denoted $t|_p$ is defined by:

- $t|_\epsilon := t$
- $t|_{ip'} := t_i|_{p'}$, if $t = f(t_1, \ldots, t_n)$

For $p \in Pos(t)$, the term $t[s]_p$ is created by replacing the subterm at position p with s. In other words:

- $t[s]_\epsilon := s$
- $f(t_1, \ldots, t_n)[s]_{ip'} := f(t_1, \ldots, t_i[s]_{p'}, \ldots, t_n)$

This can be extended to a set $I \subset Pos(t)$ of parallel positions, so $t[s]_I$ replaces each term at each $i \in I$ with s. An *equation* is an ordered-pair of terms s and t, written $s \approx t$. Given a set E of equations, new equations can be derived using the rules of *equational logic* as follows:

$$\vdash s \approx s \text{ (reflexive)}$$
$$s \approx t \vdash t \approx s \text{ (symmetric)}$$
$$s \approx t, t \approx u \vdash s \approx u \text{ (transitive)}$$
$$s_1 \approx t_1, \ldots, s_k \approx t_k \vdash f(s_1, \ldots, s_k) \approx f(t_1, \ldots, t_k) \text{ (congruence)}$$
$$s \approx t \vdash s\sigma \approx t\sigma \text{ (substitution)}$$

Let $Th(E)$ be the set of equations, $s \approx t$, such that $E \vdash s \approx t$. Moreover, let $Th_G(E)$ be the set of ground equations in $Th(E)$. We say two sets of equations, E and E', are *ground-equivalent* (written $E \equiv_G E'$) if $Th_G(E) = Th_G(E')$. A presentation of T is any finite set E of equations such that $Th(E) = T$.

Alternatively, we say that $s \approx_e^p t$ if there is a substitution σ, an equation $e := l \approx r$, and a position $p \in Pos(s)$ such that $s|_p = l\sigma$ and $s[r\sigma]_p = t$. The equation $s \approx_e t$ denotes that there is a $p \in Pos(s)$ such that $s \approx_e^p t$. We can think of this as replacing the subterm $l\sigma$ at position p in s with $r\sigma$. We write $s \approx_E t$ if there is a finite sequence of equations and positions $(e_0, p_0), \ldots, (e_k, p_k)$ such that $s = v_0 \approx_{e_0}^{p_0} v_1 \approx_{e_1}^{p_1} \cdots \approx_{e_k}^{p_k} v_{k+1} = t$. It holds that $s \approx_E t$ if and only if $E \vdash s \approx t$.

For example, given the presentation $E := \{e_1 := f(x) \approx f(y), e_2 := g(f(a)) \approx g(b)\}$, we can prove $g(f(b)) \approx g(b)$ by the derivation $g(f(b)) \approx_{e_1}^1 g(f(a)) \approx_{e_2}^\epsilon g(b)$. Given the presentation $E := \{e_1 := f(x, y) \approx f(x, x), e_2 := f(x, y) \approx f(y, x)\}$, we can prove $f(f(a, b), f(b, a)) \approx f(f(b, a), b)$ by the derivation $f(f(a, b), f(b, a)) \approx_{e_2}^1 f(f(b, a), f(b, a)) \approx_{e_1}^\epsilon f(f(b, a), b)$.

Let $EQ(E)$ denote the set of equivalence classes induced by E. We use $[t]_E$ to denote the equivalence class of E containing the term t and $[t]$ when E is implicitly known. A *ground equivalence class* contains at least one ground term, and $EQ_G(E)$ is the set of ground equivalence classes of E.

We use $Vars(\cdot)$ to denote the set of variables occurring in any object, such as a term, equation, or set of equations.

We define the subterms of a term recursively by:

$$Subterms(g(s_1, \ldots, s_k)) := \{g(s_1, \ldots, s_k)\} \cup \bigcup_i Subterms(s_i)$$

We lift the definition to sets S of terms:

$$Subterms(S) := \bigcup_{s \in S} Subterms(s)$$

We say that a set S of terms is *subterm-closed* if $Subterms(S) = S$. We say that s *appears* in t (resp. $t \approx u$) if $s \in Subterms(t)$ (resp. $Subterms(t) \cup Subterms(u)$)

The size of a term is defined by $\|\cdot\|$ so that $\|f(s_1, \ldots, s_k)\| := 1 + \Sigma_i \|s_i\|$ and $\|a\| := 1$ for all symbols f of arity k, constants a, and terms $s_1, \ldots, s_k$. This can be extended to equations, sets of terms, and sets of equations in the natural way.

The *depth* of a term t is the length of the largest $p \in Pos(t)$. A term s (resp. equivalence class c) *occurs* in t at depth d if there is a $p \in Pos(t)$ of length d such that $t|_p = s$ (resp. $s \in C$ such that $t|_p = s$). A term t is *shallow* if no variable occurs at a depth greater than or equal to 2. An equation $s \approx t$ is shallow if s and t are shallow, and a set of equations is shallow if all of its equations are shallow. For example, $f(h(h(a)), x, b) \approx x$ and $g(x, y, b) \approx h(x)$ are shallow, while $f(h(x), a) \approx x$ and $h(h(x)) \approx h(x)$ are not. A theory T is shallow if there exists a shallow presentation E such that $Th(E) = T$.

An equation $s \approx t$ is collapsing if $t \in V$ and $t \in Subterms(s)$. A set of equations is collapsing if it contains at least one collapsing equation, and any theory is collapsing if each of it's presentations is collapsing. Any equation, set of equations, or theory is non-collapsing if it is not collapsing. Non-collapsing shallow theories can be presented by equations where variables only appear at depth 1.

For any equivalence class c, and any term s, we use $D1P_c(s)$ to denote the set of depth 1 positions of terms from c in s. In other words, $D1P_c(s)$ is the largest subset of $\mathbb{N}$ such that for each $i \in D1P_c(s)$, $s|_i \in c$. Define $D1P_x(s)$ analogously for any variable, x. Let $D1_E(s)$ be the set of E equivalence classes that appear at depth 1 in s. We define $D1_E(s \approx t) := D1_E(s) \cup D1_E(t)$ and $D1_E(E') := \bigcup_{s \approx t \in E'} D1_E(s \approx t)$ We drop the subscript E when it is clear from context.

3.3 Properties of Non-Collapsing Shallow Theories

This section investigates some properties of non-collapsing shallow theories that will be useful in the following sections. We introduce a representation for non-collapsing shallow theories known as *maximally-generalized signature equations*. We will show that this representation is canonical for ground-equivalent theories up to a renaming of variables. This representation can be used to determine a canonical presentation for non-collapsing shallow theories. Assuming a fixed signature, we will also show that the size of this representation is polynomial in the size of any ground-equivalent presentation.

Throughout this section, unless otherwise stated, we will assume that every theory is non-trivial. This means that there are at least two ground terms that are equivalent and at least two ground terms that are not equivalent in the target theories.

Signature Equations

Given an equational theory E, a *signature*, *sig* (defined over E), consists of a function symbol $f \in \Sigma_k$ and a set of equivalence classes $C_1, \ldots, C_k \in EQ_G(E)$, represented $\langle f, C_1, \ldots, C_2 \rangle$. We call f the *head* and $C_1, \ldots, C_k$ the *body* of the signature.We write $sig[i]$ for each C_i.

An *instance* of *sig* is a term $f(s_1, \ldots, s_k)$, where for each i, $s_i \in C_i$. We write $Inst(sig)$ to represent the set of instances of sig. We may also represent *sig* by $\langle f, s_1, \ldots, s_k \rangle$ when $f(s_1, \ldots, s_k) \in Inst(sig)$, since each s_i acts as a representative element of C.

An *extended signature*, sig' is defined analogously, but may contain variables in place of equivalence classes. Moreover, instances of extended signatures may replace like variables with like terms. When an instance contains all the same variables as its signature, it is a maximall-generalized instance For example $\langle f, a, x, x \rangle$ and $\langle f, [a]_E, x, x \rangle$ are both representations of the same extended signature, and $f(a, x, x)$ and $f(a, b, b)$ are instances of this signature, though $f(a, x, x)$ is the only maximally-generalized instance. Given two signatures, sig_1 and sig_2, we define the order $\preceq$ such that $sig_1 \preceq sig_2$ if $Inst(()sig_1) \subset Inst(()sig_2)$. We say $sig_1 \prec sig_2$ if $sig_1 \prec sig_2$ but $sig_2 \not\prec sig_1$. For example, $\langle f, a, b, b \rangle \prec \langle f, a, x, x \rangle$. Likewise, $\langle f, a, x, b \rangle \not\prec \langle g, a, x, b \rangle$ since they have different heads, and $\langle f, a, x, y \rangle \not\prec \langle f, x, x, y \rangle$. If $sig_1 \prec sig_2$, we say that sig_2 is more general than sig_1. For a set I of indices, we write $sig[c]_I$ to replace each element at each $i \in I$ with c.

A *signature equation* is an pair of extended signatures, written $sig_1 \approx sig_2$ for signatures sig_1 and sig_2. An *instance* of a signature equation $sig_1 \approx sig_2$ is an equation $s_1 \approx s_2$, where s_1 is an instance of sig_1 and is an instance s_2 of sig_2. The signature equation $sig_1 \approx sig_2$ (defined over E) *holds* on E if for every ground instance $s_1 \approx s_2$ of $sig_1 \approx sig_2$, $s_1 \approx_E s_2$. A *maximally generalized* signature equation (MGSE) is a signature equation $sig_1 \approx sig_2$ defined over E such that for all signatures sig_1' and sig_2' such that $sig_1 \preceq sig_1'$ and $sig_2 \preceq sig_2'$, there is an instance $s_1' \approx s_2'$ of the equation such that $s_1' \neq_E s_2'$. The set of maximally generalized signature equations of a theory E is written $MGSE(E)$. Every pair of ground terms equivalent in E is an instance of some MGSE of E. Note that for any presentation E', it holds that $E \equiv_G E'$ if and only if $MGSE(E) = MGSE(E')$ (up to a renaming of variables). Therefore, we can try to learn an equational theory by learning its MGSEs.

This will be accomplished by learning the equivalence classes that appear in MGSEs. We say an equivalence class is *essential* if appears in the body of some MGSE. An essential term is a term belonging to an essential class. We will use $\mathcal{EC}(E)$ to denote the set of of essential classes in E, and will use use $\mathcal{EC}$ when E is clear from context.

Example 2. *Let $E := \{f(a, x) \approx b\}$. This yields MGSE $\langle f, a, x \rangle \approx \langle b \rangle$. The equivalence class $[a]$ is an essential class, since it appears in the body of $\langle f, a, x \rangle$ but $[b]$ is not because it appears in the head of $\langle b \rangle$.*

Example 3. *Let $E := \{f(x, y) \approx f(y, x), f(a, b) = c\}$. This yields MGSEs $\langle f, a, b \rangle \approx \langle c \rangle$, $\langle f, b, a \rangle \approx \langle c \rangle$, and $\langle f, x, y \rangle \approx \langle f, y, x \rangle$. The classes $[a]$ and $[b]$ are essential.*

The Canonical Representation

We will show how MGSEs and signature equations can be used to construct canonical representations of non-collapsing shallow theories. The learning algorithm will learn these representations. Bounding the size of these representations will allow us to bound the complexity of our learning algorithms.

Lemma 1. *Given a ground term u, $[u]$ is an essential term if any only if it appears in the body of some $MGSE$, $sig_1 \approx sig_2$.*

Proof. Assume $[u]$ is an essential term, and let $s \approx t$ be the ground equation and v be the ground term such that $s[v]_I \not\approx t[v]_J$, where $I := D1P_{[u]}(s)$ and $J := D1P_{[u]}(t)$. Let $sig_1 \approx sig_2$ be a signature equation with $s \approx t$ as an instance. Assume for contradiction that $[u]$ is not in the body of $sig_1 \approx sig_2$. So there must only be variables at position I in sig_1 and position J in sig_2. Let x be any such variable. Assume for contradiction w.l.o.g. that x also appears in sig_1 at some position i not in I. Since $i \notin I$, $u \not\approx_E s|_i$, so $s \approx t \notin Inst(sig_1 \approx sig_2)$. Therefore $D1P_x(s) \subseteq I$ and $D1P_x(t) \subseteq J$, by contradiction. Therefore, $s[v]_I \approx t[v]_J \in Inst(sig_1 \approx sig_2)$. By contradiction, $[u]$ must be in the body of $sig_1 \approx sig_2$.

Now assume there is an MGSE, $sig_1 \approx sig_2$, with $[u]$ in it's body at positions I in sig_1 and J in sig_2. Let $s \approx t$ be any instance of $sig_1 \approx sig_2$ such that $D1P_{[u]}(s) = I$ and $D1P_{[u]}(t)$. Assume for contradiction that for all v, $s[v]_I \approx_E t[v]_J$. Then every ground instance of $s[x]_I \approx t[x]_J$ holds for E. Since the choice of $s \approx t$ was arbitrary, we can replace each $[u]$ in $sig_1 \approx sig_2$ with x, for any variable x not appearing in $sig_1 \approx sig_2$, to get a new signature that holds for E. Thus $sig_1 \approx sig_2$ is not an MGSE. By contradiction, there must be a v such that $s[v]_I \not\approx_E t[v]_J$ and $[u]$ is essential. □

Let $<$ be any total ordering on terms such that for any terms s and t, $\|s\| < \|t\|$ implies $s<t$. For each $C \in EQ_G(E)$, let the *representative term* of C, called rep_C, be minimal ground term in C with respect to $<$. Let $s \approx t$ be an equation with variables and representative terms at depth 1. We say that $s \approx t$ is a *representative equation* of E if it contains only variables and representative terms of essential classes at depth 1 and every ground instance of $s \approx t$ is provable by E. The *representative presentation* E_{rep} is the set of representative equations of E. For a fixed $<$, we can see that E_{rep} is canonical up to a renaming of variables. The following proposition shows that E_{rep} is in fact a presentation of E.

Proposition 1. *Given the presentation E_{rep} constructed from E as above with ordering $<$, $E_{rep} \equiv_G E$.*

Proof. It is easy to check that all ground terms that are equivalent in E_{rep} are equivalent in E, by the definition of E_{rep}. To show the other direction, we will show by induction on k that for all ground terms s and t such that $\|s\|, \|t\| \leq k$, $s \approx_{E_{rep}} t$ if $s \approx_E t$. Base, $k = 2$: Let $a, b \in \Sigma_0$ be distinct constants. If $a \approx_E b$, then $\langle a \rangle \approx \langle b \rangle \in MGSE(E)$, so $a \approx b \in E_{rep}$. Inductive step: Assume for all ground terms s and t such that $\|s\|, \|t\| \leq k$, $s \approx_E t$ if and only if $s \approx_{E_{rep}} t$. Let $s := f(s_1, \ldots, s_l)$ and $t := g(t_1, \ldots, t_r)$ be terms such that $\|s\|, \|t\| \leq k + 1$. If s and t have the same signature in E, then $f = g$, $l = r$, and for all i, $s_i \approx_E t_i$, so $s \approx_{E_{rep}} t$ by the inductive hypothesis. Otherwise, $s \approx t$

is an instance of an $MGSE$, $sig_1 \approx sig_2$, with representative equation $u \approx v$ in E_{rep}. For each class C such that $C = sig_1[i]$ (resp. $C = sig_2[i]$), the inductive hypothesis implies that $s|_i \approx_{E_{rep}} rep_C$ (resp. $t|_i \approx_{E_{rep}} rep_C$), since $\|rep_C\| \leq \|s|_i\| \leq k+1$ (resp. $\|rep_C\| \leq \|t|_i\| \leq k+1$) by the definition of rep_C and $<$. For each variable x appearing at positions $I \subseteq \mathbb{N}$ in sig_1 and $I' \subseteq \mathbb{N}$ in sig_2, we can choose some $i \in I$ (resp. $i' \in I'$) and let $rep_x := s_i$ (resp. $rep_x := t_{i'}$). Using the inductive hypothesis, we can see that $\forall j \in I$, $s_j \approx_{E_{rep}} rep_x$ and $\forall j \in I'$, $t_j \approx_{E_{rep}} rep_x$. Therefore, we can construct terms $s' := f(s'_1, \ldots, s'_l)$ and $t' := g(t'_1, \ldots, t'_r)$ such that for all j, if $sig_1[j]$ (resp. $sig_2[j]$) is a class C, then $s'_i := rep_C$ (resp. $t'_i := rep_C$) and if $sig_1[j]$ (resp. $sig_2[j]$) is a variable x, then $s'_i := rep_x$ (resp. $t'_i := rep_x$). By this construction, we can see that $s \approx_{E_{rep}} s'$, $t \approx_{E_{rep}} t'$, and $s' \approx_{u \approx v} t'$. Since $u \approx v \in E_{rep}$, this means that $s \approx_{E_{rep}} t$. Therefore, $E_{rep} \equiv_G E$. □

The following lemma shows that terms that do not appear at depth 1 in a presentation can be replaced with variables.

Lemma 2. *Let E be any non-collapsing shallow presentation and let s, t be terms in $T(\Sigma, V)$ such that $s \approx_E t$. Assume there is a $u \in T(\Sigma)$ such that $[u] \notin D1(E)$. Let $I := D1P_{[u]}(s)$ and $J := D1P_{[u]}(t)$. Then $s[x]_I \approx_E t[x]_J$ for any $x \in V \backslash (Vars(s) \cup Vars(t))$.*

Proof. Assume $s = v_0 \approx^{p_0}_{e_0} v_1 \approx^{p_1}_{e_1} \ldots v_{r-1} \approx^{p_{r-1}}_{e_{r-1}} v_r = t$. We will prove the lemma by induction on r. Base $r = 0$: $s = t$, so $s[x]_I = t[x]_J$. Induction step: Assume $s[x]_I \approx_E v_{r-1}[x]_K$, where $K := D1P_{[u]}(v_{r-1})$. We will show that $v_{r-1}[x]_K \approx_E t[x]_J$ for all possible cases of p_{r-1}: I) If $p_{r-1} \geq k$ for some $k \in K$, then $v_{r-1}|_k \approx_E v_r|_k \in [u]$. So $K = J$ and $v_{r-1}[x]_K = t[x]_J$. II) If $p_{r-1} \geq n \in \mathbb{N} \backslash K$, then $v_{r-1}[x]_I \approx^{p_{r-1}}_{e_{r-1}} v_r t$. III) If $p = \epsilon$, then let $s' \approx t'$ equal e_{r-1}. There is a $U \subset V$ such that for each $k \in K$ and each $j \in J$, $s'|_k, t'|_j \in U$ (otherwise $s'|_k$ or $t'|_j$ is in $[u]$, violating our hypothesis). By the definition of $\approx_E$, there is a substitution $sigma$ such that $v_{r-1} = s'\sigma$ and $t = t'\sigma$. Note that this implies that no variable in U appears anywhere other than K in v_{r-1} and J in t. We can define σ' such that $\sigma'(y) := x$ for each $y \in U$ and $\sigma'(y) := \sigma(y)$ for all $y \in V \backslash U$. We then get that $s'\sigma' = v_{r-1}[x]_K$ and $t'\sigma' = t[x]_J$, so $v_{r-1}[x]_K \approx_E t[x]_J$. Thus $s[x]_I \approx_E t[x]_J$. □

The following lemma will be useful to prove a bound on the size of E_{rep} in terms of the size of any other equivalent presentation.

Lemma 3. *For any non-collapsing shallow presentation E such that $|EQ_G(E)| \geq |D1(E)| + 2d$, $\mathcal{EC}(E) \subseteq D1(E)$.*

Proof. Assume for contradiction that the lemma is not true. So $|EQ_G(E)| \geq |D1(E)| + 2d$ but $\mathcal{EC}(E) \not\subseteq D1(E)$. Let c be an essential class that does not appear at depth 1 in E. Since c is essential, is must appear in the body of some $MGSE$ of E, $sig_1 \approx sig_2$. Choose a variable x' not in $Vars(sig_1 \approx sig_2)$ and form the signature $sig'_1 \approx sig'_2$ by replacing each occurrence of c in $sig_1 \approx sig_2$ with x'. We will show that $sig'_1 \approx sig'_2$ holds for E, which implies that c is not essential. For each variable $x_i \notin Vars(sig_1 \approx sig_2)$, choose a new $c_i \in EQ_G(E) \backslash D1(E)$ and set $\sigma(x_i) := t_i$, where $t_i \in c_i$. We can choose these distinct c_i classes since $|EQ_G(E)| \geq |D1(E)| + 2d$. Take any instance $s' \approx t'$ of $sig_1\sigma \approx sig_2\sigma$. Since $s' \approx t'$ is ground and $sig_1 \approx sig_2$ holds for E,

$s' \approx_E t'$. Let $I_0 := D1P_{c'}(s')$ and $J_0 := D1P_{c'}(t')$ and set $s_0 := s'[x']_{I_0}$ and $t_0 := t'[x']_{J_0}$. Since $c' \notin D1(E)$, we can apply lemma 2 to see that $s_0 \approx_E t_0$. Now for each $x_i \in Vars(sig_1 \approx sig_2)$ set $I_i := D1P_\sigma x_i(s)$m $J_i := D1P_\sigma x_i(s)$, $s_i := s_{i-1}[x_i]_{I_i}$, and $t_i := t_{i-1}[x_i]_{J_i}$. Since σx_i is not in $D1(E)$, we can apply lemma 2 to see that $s_i \approx_E t_i$ for each i. If there are r variables in $sig_1 \approx sig_2$, then it is easy to check that $s_r \approx t_r$ is a maximally-generalized instance of $sig'_1 \approx sig'_2$, and $s_r \approx_E t_r$. Thus, $sig'_1 \approx sig'_2$ holds for E, and c is not essential by contradiction. So $\mathcal{EC}(E) \subseteq D1(E)$. □

To understand the above lemma, take for example a presentation E over the alphabet $\Sigma := \{f : 3, g : 2, a : 1, b : 1, c : 1\}$. Assume that E has $\langle f, y, a, z\rangle \approx \langle g, a, y\rangle$ as an MGSE and that $[a], [b], [c] \notin D1(E)$. We can generalize the signature equation to $\langle f, y, x', z\rangle \approx \langle g, x', y\rangle$ and consider the instance $f(b, a, c) \approx g(a, b) \in Th_G(E)$. We can then apply lemma 2 to see that $f(b, x', c) \approx_E g(x', b)$, $f(x_1, x', c) \approx_E g(x', x_1)$, and $f(x_1, x', x_2) \approx_E g(x', x_1)$. This is a maximally-generalized instance of $\langle f, y, x', z\rangle \approx \langle g, x', y\rangle$, which must hold on E. Thus $\langle f, y, a, z\rangle \approx \langle g, a, y\rangle$ must not be an MGSE, and we have a contradiction.

Theorem 1. *Let E be any non-collapsing shallow presentation over the alphabet Σ, and let E_{rep} be the representative presentation formed from the essential classes of E. Then $|E_{rep}| \leq |\Sigma|^2(2d|E| + 4d)^{2d}$.*

Proof. Only essential terms and variables can appear at depth one in E_{rep}. Therefore, since there are at most $|2d|$ variables in any equation in E_{rep} and $|\Sigma|^2$ possible pairs of root symbols, we get that $|E_{rep}| \leq |\Sigma|^2(|\mathcal{EC}(E)| + 2d)^{2d}$. It remains to be shown that $|\mathcal{EC}(E)| \leq (2d|E| + 2d)$. This proof proceeds by cases depending on the size of $EQ_G(E)$, the equivalence classes of E that contain at least one ground term. Case 1: $|EQ_G(E)| < |D1(E)| + 2d$. Since all essential classes contain at least one ground term, $\mathcal{EC}(E) \subseteq EQ_G(E)$. Since at most $2d$ classes can appear at depth 1 per equation in E, $|D1(E)| \leq 2d|E|$. Thus $|\mathcal{EC}(E)| \leq |EQ_G(E)| < |D1(E)| + 2d \leq 2d|E| + 2d$. Case 2: $|EQ_G(E)| \geq |D1(E)| + 2d$. By lemma 3, $\mathcal{EC}(E) \subseteq D1(E)$, so $|\mathcal{EC}(E)| \leq |D1(E)|$. Since $|D1(E)| \leq 2d|E|$, we get that $|\mathcal{EC}(E)| \leq 2d|E|$. □

Note that it is not possible to bound $|E_{rep}|$ in terms of $\|E\|$. For example, for any positive integer k, let $E := \{f^k(a) \approx a\}$, where $f^k(a)$ is the result of applying f to a k times. Then $|E| = |E_{rep}| = 1$, but $\|E\| \geq k$ for any k. We can also prove a polynomial bound on $\|E_{rep}\|$ depending on $\|E\|$ and $|E|$ for any $E \equiv_G E_{rep}$.

In the following, given a complexity class c of E let $minsize_E(c)$ represent the value $\|t\|$ that is minimized for all $t \in c$.

Theorem 2. *Let E be any non-collapsing shallow presentation over the alphabet Σ, and let E_{rep} be the representative presentation formed from the essential classes of E. Then $\|E_{rep}\| \leq |E_{rep}|(\|E\|(2d)^{2d} + 2)$.*

Proof. Assume $|D1(E)|+2d \leq EQ_G(E)$. By lemma 3, $\mathcal{EC}(E) \subseteq D1(E)$. Let $m' := max_{c\in\mathcal{EC}(E)}\{minsize_E(c)\}$. Since a term from every essential class must appear in E, we get that $m' \leq \|E\|$. Each equation

in E_{rep} contains only variables and essential terms at depth 1, and all essential terms are the smallest in their equivalence class. Therefore, for each $s \approx t \in E_{rep}$, $\|s \approx t\| \leq 2dm' + 2$. So $\|E_{rep}\| \leq |E_{rep}|(2dm' + 2) \leq |E_{rep}|(\|E\|2d + 2)$

Assume $|D1(E)|+2d > EQ_G(E)$ and let $\hat{C} := EQ_G(E) \backslash D1(E)$. If $maxarg_{c \in \mathcal{EC}(E)}\{minsize_E(c)\} \in D1(E)$, then we can apply the same reasoning above to bound $\|E_{rep}\|$. Otherwise, number the c_i in $\hat{C}$ such that $minsize_E(c_1) \leq minsize_E(c_2) \leq \ldots$, and let $m := max_{c \in D1(E)}\{minsize_E(c)\}$ (note $m \leq \|E\|$). The value of $minsize_E(c_1)$ is maximized if its minimal size term contains only terms of size m and has maximum arity. Therefore, $minsize_E(c_1) \leq dm$. Likewise, for each $1 \leq i \leq |\hat{C}|$, $minsize_E(c_i) \leq d \cdot minsize_E(c_{i-1})$. So $minsize_E(c_{|\hat{C}|}) \leq d^{|\hat{C}|}m \leq d^{2d-1}\|E\|$. So $\|E_{rep}\| \leq |E_{rep}|(2d^{2d}\|E\| + 2)$. □

3.4 Learning in the Limit

This section investigates the possibility of learning non-collapsing shallow theories in the limit from ground examples.

In the *learning in the limit* model, first presented in [28], the learner is trying to learn a target concept L from a concept class $C \subset 2^X$ for some space X of elements. The learner is given a sequence of examples $e_1, e_2, \ldots$. When learning from *positive examples*, these examples are drawn from the target set (i.e., *language*) L, with the guarantee that every $x \in L$ will eventually be seen in some example. When learning from *negative examples*, each e_i is of the form (x_i, b_i) where the b_i indicates whether x_i is in L, and every $x \in X$ will eventually be seen in some example.

After each new example e_t, the learner is asked to give a hypothesis L_t. It is said to learn in the limit if the learner eventually converges to the target concept, L.

In this section, X is the set $T(\Sigma)$ of all ground equations, and the concepts L are theories over ground terms.

Learning from Positive Examples

Learning ground equational presentations in the limit from positive examples is easy. After seeing some set of positive ground examples E, simply use E as the hypothesis presentation.

However, when non-ground equations are allowed, even very simple classes of equational theories are no longer learnable from positive examples. A simple corollary of Gold's theorem shows that the set of equational theories that can be presented by ground equations and (optionally) the equation $f(x, y) \approx f(y, x)$ is not learnable from positive examples. Since these are all non-collapsing shallow theories, this implies that the entire class of non-collapsing shallow theories is not learnable from positive examples.

We restate Gold's theorem below [28]:

Theorem 3. *Let $C' := \{L_\infty, L_0, L_1, \ldots\}$ be a set of formal languages such that for all i, $L_i \subset L_{i+1}$ and $L_\infty := \bigcup_i L_i$. Then no class C containing C' is learnable from positive examples.*

We will now construct a set C' of non-collapsing shallow theories that fits the above conditions. Let $\Sigma := \{a : 0, b : 0, f : 2\}$. For each $i \in \mathcal{N}$, let $E_i := \{f(s,t) \approx f(t,s) \mid depth(s), depth(t) \leq i\}$ and let $L_i := Th_G(E_i)$. Let $L_\infty := Th_G(\{f(x,y) \approx f(y,x)\})$. We can see that $C' := \{L_\infty, L_0, L_1, \dots\}$ satisfies the conditions of the above theorem. Therefore, no class whose theories can be presented using only ground equations and the equation $f(x,y) \approx f(y,x)$ can be learned from positive examples.

Learning from Positive and Negative Examples

This subsection presents an algorithm for learning non-collapsing shallow theories in the limit from examples of positive and negative ground-equations. The algorithm takes a set S^+ of positive examples and S^- of negative examples and returns a hypothesis in time polynomial in $\|S^+\| + \|S^-\|$. The hypothesis is consistent with all examples. As more examples are added, the algorithm will eventually converge on the presentation E_{rep} defined with respect to an ordering $<$.

It is fairly easy to create an algorithm that creates a consistent hypothesis in polynomial time and learns the theory in the limit. Say enumerate all non-collapsing shallow presentations, $E_1, E_2, \dots$, in order of increasing presentation size, $\|E_i\|$. Given some examples (S^+, S^-) such that $|S^+| + |S^-| = n$, the algorithm can check if there is an $i < n$ such that E_i is consistent with (S^+, S^-). If so, the algorithm returns the first such E_i. If not, the algorithm returns S^+. To check if E_i is consistent with (S^+, S^-), the learner checks that $s \approx_{E_i} t$ for each $s \approx t \in S^+$ and it checks that $s \not\approx_{E_i} t$ for each $s \approx t \in S^-$. This takes polynomial time, since the size of each E_i for $i < n$ is less than polynomial in n and checking provability of any $s \approx t$ in a non-collapsing theory takes polynomial time. Therefore, finding a consistent hypothesis takes polynomial time in the input size.

However, this algorithm is far from practical. There are at $\Omega(2^n)$ presentations of size less than n. So, this learning process will require exponentially many examples to learn most theories. The rest of this sections presents an algorithm that requires polynomially many good examples before converging on a solution.

Let E be a non-collapsing shallow presentation of the target theory defined over the alphabet Σ. Let $<$ be an ordering on terms such that for all terms s and t, $\|s\| < \|t\|$ implies $s < t$. Given the examples (S^+, S^-), the hypothesis $\hat{E}$ is constructed as follows:

- A set A of essential terms is found. (This is described in more detail in the next subsection)
- The algorithm creates a set B of non-collapsing shallow equations with only terms from A and variables at depth 1.
- For each equation $e \in B$, the algorithm checks whether $\hat{E} \cup \{e\} \cup S^+ \vdash u \approx v$ for any $u \approx v \in S^-$. If not, e is added to $\hat{E}$.
- For all $e, e' \in \hat{E}$ such that $e \prec e'$, e is removed from $\hat{E}$
- The algorithm returns the hypothesis $\hat{E} := \hat{E} \cup \{s \approx t \in S^+ \mid s \not\approx_{\hat{E}} t\}$

This algorithm takes inspiration from the RPNI algorithm for learning regular languages from positive and negative examples [45]. In particular, both algorithms try to generalize as much as possible without contradicting the known negative examples.

Identifying Essential Terms

We say that the terms s and t are *provably distinct* if there is a $u \approx v \in S^-$ such that $s \approx_{S^+} u$ and $t \approx_{S^+} v$.

We say a ground signature $f(s_1, \ldots, s_k) \approx g(s_{k+1}, \ldots, s_{k+r})$ is *classified* if for every s_i and s_j, either $s_i \approx_{S^+} s_j$ or s_i is provably distinct from s_j.

Let $s \approx t$ be a classified equation such that $s \approx_{S^+} t$, and let u be a term with $I := D1P_{[u]}(s)$ and $J := D1P_{[u]}(t)$. If there is a v such that $s[v]_I \approx t[v]_J$ is classified and $s[v]_I$ is provably distinct from $t[v]_J$, then u is an essential term by the definition of essential classes.

By finding pairs of equations $s \approx t$ and $s[v]_I \approx t[v]_J$ as above, the algorithm assembles a set A' of essential terms. Since A' might contain multiple terms from the same equivalence class, a new set A is constructed of provably distinct terms is constructed as follows: Set $A := \emptyset$. In increasing order of $<$, take each $s \in A'$. If s is provably distinct from each element of A, then add s to A'.

Characteristic Samples

A *characteristic sample* is a pair of sets (S'^+, S'^-) such that whenever $S'^+ \subseteq S^+$ and $S'^- \subseteq S^-$, the algorithm will yield a correct hypothesis. We will show that the algorithm admits a characteristic sample for every non-collapsing shallow theory. This implies that the algorithm will learn non-collapsing shallow theories in the limit.

Note that the algorithm will always yield the correct solution if there is only one equivalence class.Therefore, we can assume that there are at least two equivalence classes in the target theory.

We will now construct the characteristic sample (S'^+, S'^-) for the non-collapsing shallow theory $Th_G(E)$ with order $<$. As we describe the construction, we will show that any sample containing the characteristic sample will cause the algorithm to yield the hypothesis E_{rep}.

For each essential class C, recall that rep_C is the minimal element of C with respect to $<$.

For each rep_C, find an MGSE, $sig_1 \approx sig_2$, C in it's body and let I and J be the positions of C in sig_1 and sig_2, respectively. For a $v \notin C$, find a pair of equations $s \approx t$ and $s[v]_I \approx t[v]_J$ such that $s \approx t \in Inst(sig_1 \approx sig_2)$ and $s[v]_I \not\approx_E t[v]_J$. Add $s \approx t$ to S^+ and add $s[v]_I \approx t[v]_J$ to S^-. This guarantees that the algorithm will add rep_C to A'.

Add the set $\{rep_C \approx rep_{C'} \mid C, C' \in \mathcal{EC}(E)\}$ to S'^-. This guarantees that the set A will contain all rep_C terms for each $C \in \mathcal{EC}(E)$. No other essential terms will added to A, since they must be in some $C \in \mathcal{EC}(E)$ and thus cannot be provably distinct from rep_C. Therefore, the algorithm will find the set $A := \{rep_C \mid C \in \mathcal{EC}(E)\}$.

For each signature equation $sig_1 \approx sig_2$ that has only variables and essential classes in its body and does not hold for E, find an equation $s \approx t \in Inst(sig_1 \approx sig_2)$ such that $s \not\approx_E t$ and add $s \approx t$ to S^-. Thus, the representative equation for $sig_1 \approx sig_2$ will not be added to $\hat{E}$.

Every representative equation in E_{rep} will be added to $\hat{E}$, since there can be no equation in S^- to contradict it. By the definition of E_{rep}, all other equations that hold for E are instances of E_{rep}. Therefore, the algorithm will return E_{rep} as the final hypothesis.

Examples

Example 4. *Let* $\Sigma := \{f : 1, a : 0\}$, $S^+ := \emptyset$, *and* $S^- := \{f(a) \approx a\}$. *There are no classified equations, so no essential terms are identified. The algorithm tries to add* $f(x) \approx a$, *but fails since it can be used to prove* $f(a) \approx a$. *It then tries* $f(x) \approx f(y)$ *and succeeds. The hypothesis presentation is therefore* $\hat{E} := \{f(x) \approx f(y)\}$.

Example 5. *Let* $\Sigma := \{f : 1, a : 0, b : 0\}$, $S^+ := \{f(a) \approx f(b)\}$, *and* $S^- := \{a \approx b, f(f(a)) \approx f(a), f(f(a)) \approx f(b), f(a) \approx b, f(a) \approx a\}$. *The equations,* $f(a) \approx f(b)$, $f(f(a)) \approx f(b)$, *and* $f(a) \approx f(f(a))$ *are classified. They are used to show that both* a *and* b *are essential terms, since* $f(a) \not\approx_E a$ *and* $f(a) \not\approx_E b$. *These terms* a *and* b *are provably distinct. Thus, the algorithm tries to add the following equations:* $f(a) \approx f(b)$ *(yes),* $f(a) \approx a$ *(no),* $f(a) \approx b$ *(no),* $f(a) \approx f(x)$ *(no),* $f(b) \approx a$ *(no),* $f(b) \approx b$ *(no),* $f(b) \approx f(x)$ *(no),* $f(x) \approx f(y)$ *(no). The hypothesis presentation is therefore* $\hat{E} := \{f(a) \approx f(b)\}$.

3.5 Learning From Queries and Counter-Examples

This section presents the main result for this chapter: an efficient algorithm to learn non-collapsing shallow equational theories from ground queries and counter-examples to an oracle. These queries take the following two forms:

- *Membership Query:* The algorithm presents a ground equation $s \approx t$ to the oracle and the oracle states whether or not $s \approx t \in Th_G(E)$,
- *Equivalence Query:* The algorithm presents a hypothesis presentation E' to the oracle and the oracle states whether or not $E' \equiv_G E$. If not, the oracle also returns a counter-example $s \approx t$ from the set $(Th_G(E) \backslash Th_G(E')) \cup (Th_G(E') \backslash Th_G(E))$.

This algorithm will specifically learn the canonical presentation E_{rep} of a theory E over a fixed alphabet Σ.

Throughout this section, unless otherwise stated, we will assume that every theory is non-trivial. The algorithm can query the presentations $\emptyset$ and $\{x \approx y\}$ to the oracle in order to rule out the trivial cases.

We will first show how to learn using a set of "auxiliary symbols", $C_\alpha := \{\alpha_1, \alpha_2, \ldots, \alpha_{2d}\}$, where d is the maximum arity of any symbol in Σ. The symbols of C_α are all constants. For each $\alpha_i \in C_\alpha$, we have the guarantee that $[\alpha_i]$ is not essential.Later, we will see how to learn non-collapsing shallow theories without assuming such a set C_α exists.

The algorithm runs in iterations, creating a new hypothesis at each iteration. The structure of each iteration is as follows:

1. Use essential classes and C_α to find all MGSEs of E
2. Find set $rep_{\mathcal{EC}}$ of minimal terms from each essential class and find a hypothesis $\hat{E}$ for E_{rep}
3. Pose $\hat{E}$ as an equivalence query to the oracle.
   - If the oracle returns *true* return $\hat{E}$
   - Otherwise, we receive a counter-example $s \approx t$ such that $s \approx_E t$, but $s \not\approx_{\hat{E}} t$.
4. Use the counter-example to find an equation $s' \approx t'$ that is an instance of an unknown MGSE.

5. Use $s' \approx t'$ to find an unknown essential class
6. Start a new iteration from step 1

Those familiar with Angluin's algorithm [5] will notice similarities with the above algorithm, where states are analogous to essential classes and equations are analogous to transitions.

Finding the MGSEs

By the assumptions on C_α, we can use membership queries to infer the location of variables in each MGSE. For example, if the query $f(\alpha_1) \approx g(\alpha_1, a)$ holds, then we know that the signature equation $\langle f, x\rangle \approx \langle g, x, a\rangle$ holds, since otherwise $[\alpha_1]$ would be essential. Given a set C of representative essential terms of E, we can query all (polynomially many) pairs of terms from the set $\{f(s_1, \ldots, s_k) \mid f \in \Sigma_k, \forall i, s_i \in C \cup C_\alpha\}$. We can find signature equations (and thus MGSEs) from the queries that return *true*.

Finding $rep_{\mathcal{EC}}$

We will show how to use the oracle to construct $rep_{\mathcal{EC}}$. For simplicity, we do not assume a fixed ordering $<$ on terms, and just find representative terms of minimal size. The ordering $<$ can be determined by the choices of representative terms.

Let S be a subterm-closed set containing the smallest known term from each essential class. We will proceed iteratively, finding a hypothesized representative $rep^j_{[s]}$ at each step j for each $s \in S$. At each iteration, we will perform the following actions on each $s \in S$ in order of increasing size. Start with $j = 1$. If s is a constant and $rep^j_{[s]}$ is not yet defined, set $rep^j_{[s]} := s$. Now assume $rep^j_{[u]}$ is defined for each $u \in S$ such that $\|u\| < \|s\|$, but $rep^j_{[s]}$ is not yet defined. Let $s := f(s_1, \ldots, s_k)$. Consider each MGSE e of the form $sig_1 \approx sig_2$ such that s is an instance of sig_1. We will construct a term t_e such that $s \approx t_e$ is an instance of $sig_1 \approx sig_2$. Let $sig_1 := \langle f, c_1, \ldots, c_k\rangle$ and $sig_2 := \langle g, d_1, \ldots, d_r\rangle$. Consider each $i \in \{1, \ldots, r\}$. If d_i is a variable and equal to some c_j in sig_1, then set $t_i := rep^j_{[s_j]}$. If d_i is a variable that doesn't appear in sig_1 then set $t_i := a$ for some constant a (choose the same constant each time). Otherwise, d_i is an essential class. If $rep^j_{d_i}$ is not yet defined, then stop constructing t_e. Otherwise, set $t_i := rep^j_{d_i}$. Let $t_e := g(t_1, \ldots, t_r)$. Set $rep^j_{[s]}$ equal to the lowest-depth t_e of all such MGSEs.

Continue this process until $rep^j_{[s]} = rep^{j+1}_{[s]}$ for all $s \in S$. By induction, it is easy to check that after each iteration j, all classes with representative elements of size less than or equal to j are assigned a min-size representative. Therefore, this process completes after $max_{s\in S}\{\|s\|\}$ iterations.

Assembling $\hat{E}$ and Handling Counter-Examples

So, assuming that we have identified all essential classes, we can efficiently find a presentation $\hat{E}$ equal to E_{rep}. If we are missing an essential class, however, then $\hat{E}$ will not correctly identify E and the oracle will return some counter-example, $s \approx t$. This counter-example will be positive, meaning that $s \approx_E t$, but $s \not\approx_{\hat{E}} t$. This is possible in the following cases, which we will handle differently:

1. $s \approx t$ is an instance of an MGSE $sig_1 \approx sig_2$, but the representative equation for this $MGSE$ cannot be applied to $s \approx t$. Either, there is an essential term u at depth 1 such that $u \not\approx_{\hat{E}} rep_{[u]}$ or there are parallel terms u and v at depth 1 such that $u \approx_E v$ but $u \not\approx_{\hat{E}} v$. In either case, we can recurse, treating $u \approx rep_{[u]}$ or $u \approx v$ as our new counter-example.
2. $s \approx t$ is not an instance of any $MGSE$, and so $s \approx t$ is an instance of a missing MGSE.

Each time a counter-example is processed in case 1, we obtain a smaller counter-example. By the construction of $\hat{E}$, if a counter-example has constants on both sides, it will already be in $\hat{E}$. Therefore, this process will always find an instance $s' \approx t'$ of a missing MGSE.

Case 2 can only occur if there is an essential class that is missing from our known set of essential classes. To find the missing essential class, take any u at depth 1 in s' or t' that is not in any known essential class (use oracle queries to confirm this). Let $I := D1P_{[u]}(s')$ and $J := D1P_{[u]}(t')$. Query $s'[\alpha_1]_I \approx t'[\alpha_1]_J$ to the oracle. By lemma 1, u is essential if and only if this query returns false. Otherwise, repeat the same process on another non-essential term.

Once the essential class is found, the algorithm begins the next iteration.

Example 6. *Let $\Sigma := \{g : 1, a : 0, b : 0, c : 0\}$ and assume that the target theory can be presented by $E := \{g(a) \approx g(b), g(b) \approx c\}$. The algorithm queries $\hat{E} := \{x \approx y\}$ to the oracle and the oracle returns false (the counter-example is ignored). The current hypothesis is that $\mathcal{EC} = \emptyset$. The algorithm creates a hypothesis for $MGSE(E)$ by querying $g(\alpha_1) \approx g(\alpha_2)$ (false), $g(\alpha_1) \approx a$ (false), $g(\alpha_1) \approx b$ (false), and $g(\alpha_1) \approx c$ (false). So the hypothesis for $MGSE(E)$ is $\emptyset$ and the hypothesis presentation $\hat{E} := \emptyset$ is passed to the oracle. The oracle returns $g(g(a)) \approx g(g(b))$ as a positive counter-example, meaning $g(g(a)) \approx_E g(g(b))$ but $g(g(a)) \not\approx_{\hat{E}} g(g(b))$. The algorithm queries all depth-1 terms in the counter-example (i.e., $g(a) \approx g(b)$ (true)) to determine that the signature of the equation with respect to E is $\langle g, [g(a)]\rangle \approx \langle g, [g(a)]\rangle$. This signature holds in $\hat{E}$, so $\hat{E}$ must fail to prove $g(a) \approx g(b)$. The equation $g(a) \approx g(b)$ is treated as the new positive counter-example and the query $a \approx b$ (false) shows that its signature is $\langle g, [a]\rangle = \langle g, [b]\rangle$. This signature equation must be an instance an unknown MGSE, so there must be an essential class in its body. To determine which classes are essential, the algorithm queries $g(\alpha_1) \approx g(b)$ (false), $g(\alpha_1) \approx g(\alpha_2)$ (false), and $g(a) \approx g(\alpha_1)$ (false). This implies that $\langle g, a\rangle \approx \langle g, b\rangle$ is an MGSE, and that $[a]$ and $[b]$ are essential classes. The hypothesis set of MGSEs is formed by making the following queries: $g(a) \approx g(b)$ (true), $g(a) \approx g(\alpha_1)$ (false), $g(\alpha_1) \approx g(b)$ (false), $g(\alpha_1) \approx g(\alpha_2)$ (false), $g(a) \approx a$ (false), $g(a) \approx b$ (false), $g(a) \approx c$ (true), $g(b) \approx a$ (false), $g(b) \approx b$ (false), and $g(b) \approx c$ (true). This yields the MGSEs $\langle g, a\rangle \approx c$, $\langle g, b\rangle \approx c$, and $\langle g, a\rangle \approx \langle g, b\rangle$. The algorithm chooses a and b as the representative for their respective equivalence classes, yielding $\hat{E} := \{g(a) \approx g(b), g(a) \approx c, g(b) \approx c\}$. This presentation is given to the oracle, which returns true, and the process completes.*

Example 7. *Let $\Sigma := \{f : 1, a : 0\}$ and assume that the target theory can be presented by $E := \{f(x) \approx f(y)\}$. The algorithm queries $\hat{E} := \{x \approx y\}$ to the oracle and the oracle returns false (the counter-example is ignored). The current hypothesis is that $\mathcal{EC} = \emptyset$. The algorithm creates a hypothesis for $MGSE(E)$ by querying $f(\alpha_1) \approx f(\alpha_2)$ (true) and $f(\alpha_1) \approx a$ (false). This yields the MGSE $\langle f, x\rangle \approx \langle f, y\rangle$ and the hypothesis presentation $\hat{E} := \{f(x) \approx f(y)\}$. This is given to the oracle, the oracle returns true, and the process completes.*

3.6 Learning From Queries Without Auxiliary Symbols

The algorithm from the previous section requires the existence of $2d$ distinct "auxiliary" symbols. However, it is likely that an oracle will only be able to answer queries over a given alphabet. Therefore, it is worthwhile to try to run the above algorithm without the use of these symbols. In this section, we accomplish this by maintaining a set T_α containing terms $t^1_\alpha, \ldots, t^{2d}_\alpha$ from distinct equivalents classes which are believed to be non-essential in E. We will first show how to update the algorithm, then show how to find new t_α terms.

Updating the algorithm

The new algorithm works the same as the old one, using the t_α terms in place of the C_α symbols to infer the location of variables and essential terms. Each time an MGSE is found, the *representative query* for that MGSE is stored. The representative query for an MGSE e is a pair $(s \approx t, \sigma)$, where σ maps variables to terms in $T\alpha$ and $s \approx t$ is the representative equation of e. Note that there is only one σ such that $s\sigma \approx t\sigma$ is queried to find e, so the representative query is unique.

Since the t_α terms may actually be essential, the learned MGSEs may be more general than the actual MGSEs. For example, let $E := \{f(a, x) = g(c)\}$, $t^1_\alpha := a$, $t^2_\alpha := b$, and $t^3_\alpha := d$. Using membership queries, the algorithm can determine that $f(t^1_\alpha, t^2_\alpha) \not\approx_E g(t^3_\alpha)$ (i.e., $f(a, b) \not\approx_E g(d)$), but $f(t^1_\alpha, t^2_\alpha) \approx_E g(c)$. This leads the algorithm to conclude that $\langle f, x, y\rangle \approx \langle g, [c]\rangle$ is an MGSE of E, since t^1_α is believed to mark the place of a variable. However, t^1_α is actually an essential term, and the learned MGSE is a generalization of the true MGSE, $\langle f, [a], y\rangle \approx \langle g, [c]\rangle$.

Therefore, when a hypothesis $\hat{E}$ is given to an oracle, the oracle may return a *negative* counter-example $s \approx t$, meaning $s \approx_{\hat{E}} t$ but $s \not\approx_E t$.

Assume such a negative counter-example $s \approx t$ is given. Using the algorithm described below to find a new term t'_α. Using Proposition 1, we can construct a proof that $s \approx_{\hat{E}} t$ of the form $s = u_0 \approx^{p_0}_{e_0} u_1 \cdots \approx^{p_{r-1}}_{e_{r-1}} u_r = t$. For each $i \in \{0, \ldots, r-1\}$, query $u_i \approx u_{i+1}$. Let i' be the smallest index such that $u_{i'} \not\approx_E u_{i'+1}$. Such an i' must exist since otherwise, $s \approx_E t$ would hold. By the construction of $\hat{E}$ in Proposition 1, the equation $e_{i'}$ used at step i' in the derivation must be the representative of some MGSE, $sig_1 \approx sig_2$. Let $(u \approx v, \sigma)$ be the representative query of $sig_1 \approx sig_2$. Since $u_i \not\approx_E u_{i+1}$, but $u_i \approx_{u\approx v} u_{i+1}$, we know $sig_1 \approx sig_2$ must be an over-generalization. Therefore, there is a variable in $sig_1 \approx sig_2$ that should be replaced at some positions with a ground term. For each variable $y \in Vars(sig_1) \cup Vars(sig_2)$, let $x\sigma_y := x\sigma$ for all $x \neq y$ and $y\sigma_y := t'_\alpha$. Query $s\sigma_y \approx t\sigma_y$. If the query returns *false*, then $y\sigma$ must be an essential term. So set $T_\alpha := (T_\alpha \backslash \{y\sigma\}) \cup \{t'_\alpha\}$ and set $\mathcal{EC} := \mathcal{EC} \cup \{[y\sigma]\}$. If no such variable is found, then t'_α must be essential. So set $\mathcal{EC} := \mathcal{EC} \cup \{t'_\alpha\}$, find a new t_α term, and repeat the above process.

Finding new t_α terms

Throughout the run of this algorithm, we maintain a set $C := \{C_1, \ldots, C_r\}$. Each C_i contains a set of terms that are equivalent in E, and $\bigcup_i C_i$ is subterm-closed. Each new t_α term is chosen as follows

1. If there is a constant c not in C, then for each i choose a $c_i \in C_i$. Query $c_i \approx c$.
   - If there is an i such that $c_i \approx_E c$, then add c to C_i and restart.
   - Otherwise, set $C := C \cup \{\{c\}\}$ and return $t_\alpha := c$
2. For each symbol f of arity k and each $(i_1, \ldots, i_{k+1}) \in \{1, \ldots, r\}^{k+1}$, query $f(c_{i_1}, \ldots, c_{i_k}) \approx c_{i_{k+1}}$, where each c_i is in C_i
   - If $f(c_{i_1}, \ldots, c_{i_k}) \approx_E c_{i_{k+1}}$, then add $f(c_{i_1}, \ldots, c_{i_k})$ to $C_{i_{k+1}}$.
   - If there is no i_{k+1} such that $f(c_{i_1}, \ldots, c_{i_k}) \approx_E c_{i_{k+1}}$, then set $C := C \cup \{f(c_{i_1}, \ldots, c_{i_k})\}$ and return $t_\alpha := f(c_{i_1}, \ldots, c_{i_k})$

As described above, the learning algorithm first finds a set of $2d$ different t_α terms, then adds at most one new t_α terms for each class in C. Therefore, $|C|$ never exceeds $2d + |\mathcal{EC}|$.

If the algorithm does not return any new t_α, then the set of classes represented in C_i is closed under each f application. Thus, every equivalence class is represented in C, and a presentation of E of size $Poly(2d + |\mathcal{EC}|)$ can be easily inferred.

Each call to this algorithm takes polynomial time assuming fixed Σ.

Example 8. *Let $\Sigma := \{g : 1, a : 0, b : 0, c : 0\}$ and assume the target theory can be presented by $E := \{g(a) \approx c\}$. The algorithm queries $\hat{E} := \{x \approx y\}$ to the oracle and the oracle returns false. It then sets $t^1_\alpha := a$ and $t^2_\alpha := b$. To find the MGSEs it queries $g(t^1_\alpha) \approx g(t^2_\alpha)$ (false), $g(t^1_\alpha) \approx a$ (false), $g(t^1_\alpha) \approx b$ (false), and $g(t^1_\alpha) \approx c$ (true). Since the algorithm believes that t^1_α (i.e., a) is not essential, the fact that $g(t^1_\alpha) \approx_E c$ leads it to conclude that $\langle g, x\rangle \approx \langle c\rangle$ is an MGSE. It passes the hypothesis $\hat{E} := \{g(x) \approx c\}$ to the oracle. The oracle returns false and gives the negative counter-example $g(g(b)) \approx c$, meaning $g(g(b)) \not\approx_E c$ but $g(g(b)) \approx_{\hat{E}} c$. This equation is provable in $\hat{E}$ by the derivation $g(g(b)) \approx_{g(x)\approx c} c$, implying that the equation $g(x) \approx c$ should not be in $\hat{E}$. Therefore, $e := \langle g, x\rangle \approx \langle c\rangle$ is an over-generalization of an actual MGSE of E. Since e was added because of the representative query $e' := g(t^1_\alpha) \approx c$, one of the t_α terms used in e' must be an essential term. The algorithm finds a new term $t'_\alpha := c$ and queries $g(t'_\alpha) \approx c$ (false). Since $g(t'_\alpha) \not\approx_E c$ and $g(t^1_\alpha) \not\approx_E c$, t^1_α must be an essential term. So the algorithm adds $[a]$ to $\mathcal{EC}$ and sets $t^1_\alpha := c$. To find the MGSEs, the algorithm queries $g(t^1_\alpha) \approx g(t^2_\alpha)$ (false), $g(t^1_\alpha) \approx a$ (false), $g(t^1_\alpha) \approx b$ (false), $g(t^1_\alpha) \approx c$ (false), $g(a) \approx a$ (false), $g(a) \approx b$ (false), and $g(a) \approx c$ (true). This yields the MGSE $\langle g, [a]\rangle \approx \langle c\rangle$. The algorithm sets a to be the representative element of its equivalence class and queries $\hat{E} := \{g(a) \approx c\}$ to the oracle. The oracle returns true and the process completes.*

Chapter 4

Modular Learning

4.1 Overview

Researchers in the field of exact active learning will often fix a concept class and then study which queries are needed to learn concepts in that class. When a different concept class is studied, an entirely new algorithm will need to be developed from scratch, even if it is composed of already studied concept classes.

This chapter studies the concept of modular learning, which breaks down concept classes into their components and study the learnability of the larger class in terms of the learnability of the components. The initial work in this area studies arbitrary boolean combinations of concepts and only considered equivalence queries [11, 14]. We study a wider range of queries, such as membership and superset queries. Moreover, we focus on the case when concepts are the cross-products of component concepts. This occurs when a system can be broken into subsystems that each act independently of each other. The subsection at the end of this introduction demonstrates these ideas in terms of learning automata, where Angluin's algorithm has has particular success in inductive synthesis [5]. When an automaton is made of several independent components, our results can reduce the number of equivalence queries exponentially in the number of components.

We will focus on the oracle queries given in Table 2.3. The results are summarized in Table 4.2, and include both upper and lower bounds. We show learning cross-products from superset queries is no more difficult than learning each individual concept. Learning cross-products from equivalence queries or subset queries is intractable, while learning from just membership queries is polynomial, though somewhat expensive. We show that when a learning algorithm is allowed to make membership queries and is give a single positive example, previously intractable problems become tractable. Finally, we discuss the computational complexity of PAC-learning and show how it can be improved when membership queries are allowed.

Introductory Example

To illustrate the learning problem, consider the sketching problem given in Figure 4.1. Here we want to find the set of possible initial values for x and y that can replace the $??$ values so that the

```
int x = ??;
int y = ??;
int a = f(x);
int b = g(y);
```

$\Phi = \phi_1(a) \wedge \phi_2(b)$

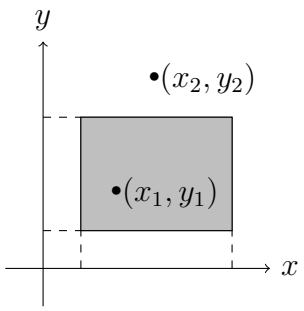

Figure 4.1: A simple partial program to be synthesized to satisfy a specification Φ (left) and the correct set of initial values for x and y (right).

program satisfies Φ, using Φ as a black-box oracle mapping x and y inputs to 'true' and 'false'.

Looking at the structure of this program and specification, we can see that the correctness of these two variables are independent of each other. Correct x values are correct independent of y and vice-versa. Therefore, the set of settings will be the cross product of the acceptable settings for each variable. If an oracle can answer queries about correct x or y values separately, then the oracle can simply learn the acceptable values separately and take their Cartesian product.

If the correct values form intervals, the correct settings will look something like the rectangle shown in Figure 4.1. An algorithm for learning this rectangle can try to simulate learning algorithms for each interval by acting as the oracle for each sublearner. For example, if both sublearners need a positive example, the learner can query the oracle for a positive example. Given the example (x_1, y_1) as shown in the figure, the learner can then pass x_1 and y_1 to the sublearners as positive examples. However, this does not apply to negative examples, such as (x_2, y_2) in the figure. In this example, x_2 is in its target interval, but y_2 is not. The learner has no way of knowing which subconcept a negative element fails on. Handling negative counterexamples is one of the main challenges of this chapter.

Sample Application: Learning Automata

The algorithms presented in Sections 4.4, 4.5, and 4.7 can be applied to the learning of finite automata. Algorithms 5 and 6 can make black-box queries to Angluin's well-known algorithm for learning finite automata [5]. Moreover, Algorithm 8 can use existing algorithms for PAC learning probabilistic finite automata [17].

Consider, for example, an automaton representing the proper input/output interactions of a simple vehicle. Say the vehicle can be split into two systems: the headlights and the motor controls.

It may be reasonable to assume that these two systems act independently. So the correct input/output sequence for motor controls is true regardless of the state of the headlights, and vice-versa. This allows us to make inferences like "The brakes should always activate after the brake pedal is pushed, regardless of whether the high-beams are on".

Now consider the problem of learning this automaton from oracle interactions or labelled data. If the labels and oracle answer are subsystem-specific (e.g., "The headlights are incorrect in this example"), then it might be possible to learn each subsystem separately.

But there are a few reasons why this subsystem-specific feedback might not be possible:

1. If the oracle answers queries by running simulations, it might not be obvious which subsystem is responsible for the fault. (e.g., "Did the car crash because the high-beams were on or because the breaks did not function correctly?")
2. When only learning from labelled examples, the data might not include this subsystem information.
3. The oracle may be implemented by an existing model which does not have this subsystem information.

This last case might be relevant in system deobfuscation, where black-box queries are made to a complex model in order to learn a simple representation of that model. That simpler representation might be used to explain the model to a human or it might be checked against a logical specification.

In particular, recent work has focused on using Angluin's algorithm to learn finite automata from RNNs [62]. If the RNN could be split into independent subsystems, but was not trained on data with subsystem-specific information, then a naive application of Angluin's algorithm would require learning the product automaton.

If the headlights-automaton has n states and the motor control automaton has m states, then the product of these two automata might be of size mn. Learning this product automaton from Angluin's algorithm would require $O(mn)$ equivalence queries. However, we can leverage the learning algorithms from this chapter to learn the same automaton in $O(m + n)$ equivalence queries. If an automaton is made of k components of size n, this process can reduce the number of equivalence queries from $O(n^k)$ to $O(kn)$, resulting in an exponential increase in efficiency.

Reusing Component Algorithms

Another important application of this work is the reuse of existing learning algorithms for previously un-studied concept classes. For example, assume we have a different model of a car, which uses an interval of acceptable internal temperatures and an automaton representing the car's motor controls. Again, we might assume the car's motor controls are correct regardless of its temperature and vice-versa. This model might be represented as the cross product of an interval of integers (representing the acceptable temperatures) and a finite automaton (representing motor controls). Although it is unlikely that a learning algorithm for this particular type of model would have been studied, individual learning algorithms for intervals and automata have been. A researcher could then use these two algorithms as the black-box sublearners to learn the entire model.

4.2 Notation

Refer back to the background section for more on concept learning. In the following proofs, we assume we are given concept classes $C_1, C_2, \ldots, C_k$ defined over instance spaces X_1, X_2,

	$\mathcal{Q}_{prod} = \mathcal{Q}_{subc}$		$\mathcal{Q}_{prod} = \mathcal{Q}_{subc} \cup \{Mem, 1Pos\}$	
$\mathcal{Q}_{subc} \downarrow$	#q		#*Mem*	#q
Pos	Not Possible		Not Possible	Not Possible
Sup	$\sum \#Sup_i$		0	$\sum \#Sup_i$
Mem	$(max_i\{\#Mem_i\})^k$		$\sum \#Mem_i$	$\sum \#Mem_i$
Sub	$k^{\sum \#Sub_i}$		$lg(k) \sum \#Sub_i$	$\sum \#Sub_i$
EQ	$k^{\sum \#EQ_i}$		$lg(k) \sum \#EQ_i$	$\sum \#EQ_i$
Mem	#*Mem*	#*EQ*	#*Mem*	#*EQ*
EQ	$(max_i\{\#Mem_i + \#EQ_i\})^k$	$\sum \#EQ_i$	$\sum \#Mem_i + lg(k) \sum \#EQ_i$	$\sum \#EQ_i$

Figure 4.2: Final collection of query complexities for learning cross products. The rows represents the set $\mathcal{Q}_{subc}$ of queries needed to learn each C_i. The columns determine the set $\mathcal{Q}_{prod}$ of queries used to learn the cross product class. In the latter case, the column is separated to also track the number of membership queries that are needed. The value k denotes the number of dimensions (i.e., concept classes) included in the cross-product. In the case when $\mathcal{Q}_{subc} = \{Mem, EQ\}$, the meaning of q is not defined, so the complexity of each case is split into $\#Mem$ and $\#EQ$.

$\ldots, X_k$. Each target concept c_i^* in each C_i is learnable from algorithm A_i (called *sublearners*) using queries to an oracle that can answer any queries in a set $\mathcal{Q}_{subc}$. This set $\mathcal{Q}_{subc}$ contains the available types of queries, which are taken from the list of queries shown in Table 2.3. For example, if $\mathcal{Q}_{subc} = \{Mem, EQ\}$, then each A_i can make membership and equivalence queries to its corresponding oracle.

For each query $q \in \mathcal{Q}_{subc}$, we say algorithm A_i makes $\#q_i$ (or $\#q_i(c_i^*)$) many q queries to the oracle in order to learn concept c_i^*, dropping the index i when unambiguous. We replace the term $\#q$ with a more specific term when the type of query is specified. For example, an algorithm A might make $\#Mem$ many membership queries to learn c.

Unless otherwise stated, we will assume any index i or j ranges over the set $\{1 \ldots k\}$. We write $\prod S_i$ or $S_1 \times \cdots \times S_k$ to refer to the k-ary Cartesian product (i.e., cross-product) of sets S_i. We use S^k to refer to $\prod_{i=1}^{k} S$.

We use vector notation $\vec{x}$ to refer to a vector of elements $(x_1, \ldots, x_k)$, $\vec{x}[i]$ to refer to x_i, and $\vec{x}[i \leftarrow x_i']$ to refer to $\vec{x}$ with x_i' replacing value x_i at position i. We define $\boxtimes_{i=1}^{k} C_i := \{\prod c_i \mid c_i \in C_i, i \in \{1, \ldots, k\}\}$. We write $\vec{c}$ or $\prod c_i$ for any element of $\boxtimes_{i=1}^{k} C_i$ and will often denote $\vec{c}$ by $(c_1, \ldots, c_k)$ in place of $\prod c_i$. The target concept will be represented as c^* or c^* which equals $(c_1^*, \ldots, c_k^*)$.

The *product oracle* is able to answer queries about the target concept c^*. The types of queries

this oracle can answer are in the set $\mathcal{Q}_{prod}$, which are taken from the queries in Table 2.3. We now have enough to state the problem of this paper.

> **Problem Statement:** For different sets of queries, $\mathcal{Q}_{subc}$ and $\mathcal{Q}_{prod}$, can we bound the number of queries needed to learn a concept in $\boxtimes C_i$ as a function of each query complexity, $\#q_i$, for each $q \in \mathcal{Q}_{subc}$?

There are far too many combinations of sets $\mathcal{Q}_{subc}$ and $\mathcal{Q}_{prod}$ to consider in one paper. In this paper we will mostly focus on the cases when $|\mathcal{Q}_{subc}| = 1$ and $\mathcal{Q}_{prod} = \mathcal{Q}_{subc}$ or $\mathcal{Q}_{prod} = \mathcal{Q}_{subc} \cup \{1Pos, \mathit{Mem}\}$, as these cases are more likely to appear in practice.

The proofs in this paper make use of the following simple observation:

Observation 1. *For sets $S_1, S_2, \ldots, S_k$ and $T_1, T_2, \ldots, T_k$, assume $\prod S_i \neq \emptyset$. Then $\prod S_i \subseteq \prod T_i$ if and only if $S_i \subseteq T_i$, for all i.*

4.3 Simple Results

We will start with a simple lower bound on learnability from *EQ*, *Sub*, and *Mem*. See Figure 4.3 for a visual representation of this proposition. We will see later that this lower bound is tight when learning from membership queries, but not equivalence or subset queries. We will also show that learning cross products from superset queries is easy.

Proposition 2. *There exists a concept C that is learnable from $\#q$ many queries posed to $\mathcal{Q}_{subc} \subseteq \{\text{Mem}, \text{EQ}, \text{Sub}\}$ such that learning C^k requires at least $(\#q)^k$ many queries when $\mathcal{Q}_{prod} = \mathcal{Q}_{subc}$.*

Proof. Let $C := \{\{j\} \mid j \in \{0 \ldots n\}\}$.

We can learn C in n membership, subset, or equivalence queries by querying $j \in c^*$, $\{j\} \subseteq c^*$, or $\{j\} = c^*$, respectively. However, a learning algorithm for C^k requires more than n^k queries. To see this, note that C^k contains all singletons in a space of size $(n+1)^k$.

So for each subset query $\{x\} \subseteq c^*$, if $\{j\} \neq c^*$, the oracle will return j as a counterexample, giving no new information. Likewise, for each equivalence query $\{j\} = c^*$, if $\{j\} \neq c^*$, the oracle can return j as a counterexample. Therefore, any learning algorithm must query $x \in c^*$, $\{x\} \subseteq c^*$, or $\{x\} = c^*$ for $(n+1)^k - 1$ values of x. □

We will now show two simple results, learning from positive examples is not always possible and learning from superset queries is easy.

Proposition 3. *There exist concepts C_1 and C_2 that are each learnable from constantly many positive queries, such that $C_1 \times C_2$ is not learnable from any number of positive queries.*

Proof. Let $C_1 := \{\{a\}, \{a, b\}\}$ and set $C_2 := \{\mathbb{N}, \mathbb{Z}\backslash\mathbb{N}\}$. To learn the set in C_1, pose two positive queries to the oracle, and return $\{a, b\}$ if and only if both a and b are given as positive examples. To learn C_2, pose one positive query to the oracle and return $\mathbb{N}$ if and only if the positive example is in $\mathbb{N}$. An adversarial oracle for $C_1 \times C_2$ could give positive examples only in the set $\{a\} \times \mathbb{N}$.

Each new example is technically distinct from previous examples, but there is no way to distinguish between the sets $\{a\} \times \mathbb{N}$ and $\{a, b\} \times \mathbb{N}$ from these examples. □

Proposition 4. *If $\mathcal{Q}_{prod} = \mathcal{Q}_{subc} = \{\text{Sup}\}$, then there is an algorithm that learns any concept $c^* \in \prod C_i$ in $\sum \#\text{Sup}_i(c_i^*)$ queries.*

Proof. Algorithm 1 learns c^* by simulating the learning of each A_i on its respective class C_i. The algorithm asks each A_i for superset queries $S_i \supseteq c_i^*$, queries the product $\prod S_i$ to the oracle, and then uses the answer to answer at least one query to some A_i. Since at least one A_i receives an answer for each oracle query, at most $\sum \#\textit{Sup}_i(c_i^*)$ queries must be made in total.

We will now show that each oracle query results in at least one answer to an A_i query (and that the answer is correct). The oracle first checks if the target concept is empty and stops if so. If no concept class contains the empty concept, this check can be skipped. At each step, the algorithm poses query $\prod S_i$ to the oracle. If the oracle returns 'yes' (meaning $\prod S_i \supseteq c^*$), then $S_i \supseteq c_i^*$ for each i by Observation 1, so the oracle answers 'yes' to each A_i. If the oracle returns 'no', it will give a counterexample $\vec{x} = (x_1, \ldots, x_k) \in c^* \setminus \prod S_i$. There must be at least one $x_i \notin S_i$ (otherwise, $\vec{x}$ would be in $\prod S_i$). So the algorithm checks $x_j \in S_j$ for all x_j until an $x_i \notin S_i$ is found. Since $\vec{x} \in c^*$, we know $x_i \in c_i^*$, so $x_i \in c_i^* \setminus S_i$, so the oracle can pass x_i as a counterexample to A_i.

Note that once A_i has output a correct hypothesis c_i, S_i will always equal c_i, so counterexamples must be taken from some $j \neq i$. □

Result: Learn $\prod C_i$ from Superset Queries
if $\emptyset \in C_i$ *for some* i **then**
  Query $\emptyset \supseteq c^*$;
  if $\emptyset \supseteq c^*$ **then**
    return $\emptyset$
for $i = 1 \ldots k$ **do**
  Set S_i to initial subset query from A_i
while *Some* A_i *has not completed* **do**
  Query $\prod S_i$ to oracle;
  if $\prod S_i \supseteq c^*$ **then**
    Answer $S_i \supseteq c_i^*$ to each A_i;
    Update each S_i to new query;
  else
    Get counterexample $\vec{x} = (x_1, \ldots, x_k)$ **for** *i = 1 … k* **do**
      if $x_i \notin S_i$ **then**
        Pass counterexample x_i to A_i;
        Update S_i to new query;
  for *i = 1 … k* **do**
    if A_i *outputs* c_i **then**
      Set $S_i := c_i$;
return $\prod c_i$;

Algorithm 1: Algorithm for learning from Subset Queries

4.4 Learning From Membership Queries and One Positive Example

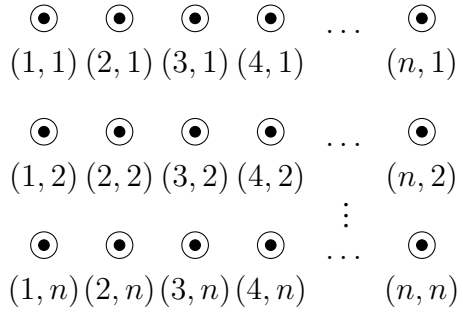

Figure 4.3: Representation for $C \times C$ in Proposition 2, when $k = 2$. The circle around each point represents a singleton set in $C \times C$.

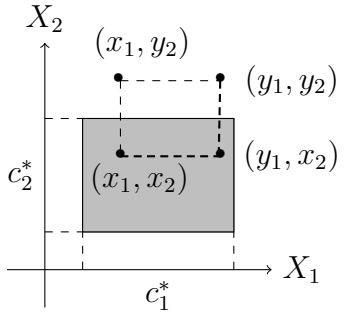

Figure 4.4: The figure for Example 9 on handling counter-examples with membership queries.

Ideally, learning the cross-product of concepts should be about as easy as learning all the individual concepts. The last section showed this is not the case when learning with equivalence, subset, or membership queries. However, when the learner is given a single positive example and allowed to make membership queries, the number of queries becomes tractable. This is due to the following simple observation.

Observation 2. *Fix sets $S_1, S_2, \ldots, S_k$, points $x_1, x_2, \ldots, x_k$ and an index i. If $x_j \in S_j$ for all $j \neq i$, then $(x_1, x_2, \ldots, x_k) \in \prod S_j$ if and only if $x_i \in S_i$.*

This suggests a simple method for handling counterexamples. Given a positive example $\vec{p} \in c^*$ and a counterexample $\vec{x} \notin c^*$, query $\vec{p}[j \leftarrow x_j] \in c^*$ for each j. Recall that $\vec{p}[j \leftarrow x_j]$ is the result of replacing the jth element of $\vec{p}$ with x_j. So by the above observation, there will be some j such that $p[j \leftarrow x_j] \notin c^*$ and we can infer that $x_j \notin c_j^*$. This process is better explained in the following example.

Example 9. *Figure 4.4 shows an example of using membership queries to handle a counter-example. The rectangle represents the target hypothesis $c^* = c_1^* \times c_2^*$. The point $\vec{p} = (x_1, x_2)$ is a positive example and (y_1, y_2) is a negative counter-example. The algorithm wants to find whether $y_1 \notin c_1^*$ or $y_2 \notin c_2^*$. It the constructs $\vec{p}[1 \leftarrow y_1]$ (i.e., (y_1, x_2)) and queries $\vec{p}[1 \leftarrow y_1] \in c^*$. The oracle returns 'true', so $y_1 \in c_1^*$. It then repeats the process and queries $\vec{p}[2 \leftarrow y_2] := (x_1, y_2) \in c^*$. The oracle returns 'false', so $y_2 \notin c_2^*$, and the oracle passes y_2 to learner A_2 as a counterexample.*

This is formalized in Algorithm 2, which shows that counterexamples can be found using logarithmically many queries.

Input: $\vec{p}$: positive example (in c^*);
$\vec{x}$: counter example;
Output: i such that $\vec{x}[i] \notin c^*_i$;
HandleCounterexample *($\vec{p}$, $\vec{x}$):*

```
// lower and upper bounds
low := 1;
up := ⌈k/2⌉ ;
for j = 1 ... lg(k) do
    p⃗' := p⃗;
    p⃗'[i] := x⃗[i] for low ≤ i ≤ up;
    size := up − low;
    Query p⃗' ∈ c*;
    if p⃗' ∉ c* then
        low := up;
    up := low + ⌈size/2⌉ ;
return low ;
```

Algorithm 2: Find dimension on which counterexample fails

Proposition 5. *Let $\vec{x}$ be a negative example. Given a positive example, Algorithm 2 returns an i such that $x_i \notin c^*_i$ using at most $lg(k)$ membership queries.*

Proof. We show by induction that at each step of the algorithm, there is an index $i \in [low, 2 \cdot up - low]$ such that $x_i \notin c^*_i$. Base: Since $up = \lceil k/2 \rceil$ and $low = 1$ the range $[low, 2 \cdot up - low]$ equals $[1, k]$. By Observation 2 there is some $x_i \notin c^*_i$, since $\vec{x} \notin c^*$. Inductive: assume this has held for all previous steps. So either there is an i in $[low, up]$ or in $[up, 2 \cdot up - low]$ such that $x_i \notin c^*_i$. By the construction of $\vec{p'}$ and Observation 2, if $i \in [low, up]$, then $\vec{p'} \in c^*$. So the new up (call it up') will be set to $low + \lceil (up - low)/2 \rceil$, so $[low, up] = [low, 2 \cdot up' - low]$ and the property still holds. If $i \notin [low, up]$, then $i \in [up, 2 \cdot up - low]$ and $\vec{p'} \notin c^*$. In this case, the new low and up (low' and up') will be up and $up + \lceil (up - low)/2 \rceil$, respectively. So $[up, 2 \cdot up - low] = [low', 2 \cdot (up') - low']$ and the property still holds. Since the size of $up - low$ decreases by one half each round, after $lg(k)$ rounds $low = up$ and so $i \in [low, 2 \cdot up - low] = [low, low]$. □

We now have enough information to present the following theorem.

Theorem 4. *In the following statements, we assume $\mathcal{Q}_{prod} = \mathcal{Q}_{subc} \cup \{\text{Mem}, 1Pos\}$.*

- *If $\mathcal{Q}_{subc} = \{\text{Mem}\}$, then c^* is learnable in $\sum \#\text{Mem}_i$ membership queries.*
- *If $\mathcal{Q}_{subc} = \{\text{EQ}\}$ (respectively $\mathcal{Q}_{subc} = \{\text{Sub}\}$), then c^* is learnable in $lg(k) \cdot \sum \#q_i(c^*_i)$ membership queries and $\sum \#\text{EQ}_i(c^*_i)$ equivalence queries (respectively $\sum \#\text{Sub}_i(c^*_i)$ subset queries).*
- *If $\mathcal{Q}_{subc} = \{\text{Mem}, \text{EQ}\}$, then c^* is learnable in $lg(k) \cdot \sum \#\text{EQ}_i(c^*_i) + \sum \#\text{Mem}_i(c^*_i)$ membership queries and $\sum \#\text{EQ}_i(c^*_i)$ equivalence queries.*

Proof. **Proof of Item 1** See Algorithm 4 below. The algorithm learns by simulating each A_i in sequence, moving on to A_{i+1} once A_i returns a hypothesis c_i. For any membership query M_i made by A_i, $M_i \in c_i^*$ if and only if $\vec{p}[i \leftarrow M_i] \in c^*$ by Observation 2. Therefore the algorithm is successfully able to simulate the oracle for each A_i, yielding a correct hypothesis c_i.

Proof of Item 2 The learning process for either subset or equivalence queries is described in Algorithm 3, with differences marked in comments. In either case, once the correct c_j is found for any j, S_j will equal c_j for all future queries, so any counterexamples must fail on an $i \neq j$.

We separately show for each type of query that a correct answer is given to at least one learner A_i for each subset (resp. equivalence) query to the cross-product oracle. Moreover, at most $lg(k)$ membership queries are made per subset (resp. equivalence) query, yielding the desired bound.

Subset Queries: For each subset query $\prod S_i \subseteq c^*$, the algorithm either returns 'yes' or gives a counterexample $\vec{x} = (x_1, \ldots, x_k) \in \prod S_i \backslash c^*$. If the algorithm returns 'yes', then by Observation 1 $S_i \subseteq c_i^*$ for all i, so the algorithm can return 'yes' to each A_i. Otherwise, $\vec{x} \notin c^*$, so there is an i such that $x_i \notin c_i^*$. Algorithm 2 is used to find the $x_i \notin c_i^*$ in $lg(k)$ queries.

Equivalence Queries: For each equivalence query $\prod S_i = c^*$, the algorithm either returns 'yes', or gives a counterexample $\vec{x} = (x_1, \ldots, x_k)$. If the algorithm returns 'yes', then a valid target concept is learned. Otherwise, either $\vec{x} \in \prod S_i \backslash c^*$ or $\vec{x} \in c^* \backslash \prod S_i$. In the second case, as with superset queries, Algorithm 2 is used to find the $x_i \notin c_i^*$ in $lg(k)$ queries. Once the $x_i \notin c_i^*$ is found it is given to A_i as a counterexample.

Proof of Item 3 The learning algorithm is described in Algorithm 5. The algorithm uses the positive example to answer membership queries. By Observation 2, for any membership query x_i made by A_i, $x_i \in c_i^*$ if and only if $\vec{p}[i \leftarrow x_i] \in c^*$. So each membership query posed by an A_i is answered with one membership query posed by the cross-product learner.

Membership queries are answered until each A_i poses an equivalence query $S_i = c_i^*$ (if A_i has terminated with the correct answer, we just assume S_i equals c_i^*). The learning algorithm then queries $\prod S_i = c^*$ and receives a counterexample $\vec{x} := (x_1, \ldots, x_k)$ or it receives a 'yes' and terminates. The algorithm checks if $\vec{x} \in \prod S_i$ and handles each case separately.

If $\vec{x} \in \prod S_i$: then $\vec{x} \in (\prod S_i) \backslash c^*$. So there is an i such that $x_i \notin c_i^*$. Algorithm 2 is used to find the $x_i \notin c_i^*$ in $lg(k)$ queries. Since $\vec{x} \in \prod S_i$, $x_i \in S_i \backslash c_i^*$, so x_i is passed to A_i as a counterexample to the query $S_i = c_i^*$. This takes at most k membership queries.

If $\vec{x} \notin \prod S_i$: then $\vec{x} \in c^* \backslash \prod S_i$, since it is a counterexample. So there is an i such that $x_i \notin S_i$. Since the algorithm has access to each S_i, it can check this explicitly without using any counter-examples.

In either case, at least one A_i receives a counterexample to its equivalence query, so this process is done at most $\sum \#EQ_i$ times, using at most k membership queries per process. This yields the stated bound on query complexity. □

for $i = 1 \ldots k$ **do**
  Set S_i to initial query from A_i
while *Some* A_i *has not completed* **do**
  Query $\prod S_i$ to oracle;
  if *The Oracle returns 'yes'* **then**
    Pass 'yes' to each A_i;
    // If $\mathcal{Q}_{subc} = \mathcal{Q}_{prod} = \{EQ\}$, each sublearner will immediately complete
  else
    Get counterexample $\vec{x} = (x_1, \ldots, x_k)$;
    if $\vec{x} \in c^* \setminus \prod S_i$ **then**
      // Only happens if $\mathcal{Q}_{subc} = \mathcal{Q}_{prod} = \{EQ\}$
      for *i = 1 . . . k* **do**
        if $x_i \notin S_i$ **then**
          Pass counterexample x_i to A_i;
          Update S_i to new query from A_i;
    else
      // Calls Algorithm 2
      Set $i' := HandleCounterexample(\vec{p}, \vec{x})$ Pass counterexample $x_{i'}$ to $A_{i'}$;
      Set $S_{i'}$ to be new query from $A_{i'}$;
Each A_i returns some c_i;
Return $\prod c_i$;

Algorithm 3: Learn when $\mathcal{Q}_{subc} = \{EQ\}$ and $\mathcal{Q}_{prod} = \{EQ, 1Pos\}$ (or $\mathcal{Q}_{subc} = \{Sub\}$ and $\mathcal{Q}_{prod} = \{Sub, 1Pos\}$)

Input: $\vec{p}$: Positive Example in X
Learn *($\vec{p}$):*
  for $i = 1 \ldots k$ **do**
    while A_i *has not completed* **do**
      Get query $x_i \in c_i^*$ from A_i;
      Query $\vec{p}[i \leftarrow x_i] \in c^*$;
      Pass answer to A_i;
      if A_i *returns guess* c_i **then**
        Break ;
  return $\prod c_i$;

Algorithm 4: Learn from Membership Queries and One Positive Example

4.5 Learning From Only Membership Queries

We have seen that learning with membership queries can be made significantly easier if a single positive example is given. If no positive example is given, then Proposition 2 gives a lower bound on the number of membership, subset, or equivalence queries needed. This section gives an algorithm showing that this bound is tight when $\mathcal{Q}_{subc} = \mathcal{Q}_{prod} = \{Mem\}$ (or $\{Mem, EQ\}$). The algorithm

uses membership queries so that either a positive example is found or the target concept is learned. Once a positive example is found, the learning algorithm from Section 4.4 can be used.

Somewhat surprisingly, even if $\mathcal{Q}_{subc} = \{Mem, EQ\}$, only membership queries are needed to find a positive example. We present the algorithm for learning when $\mathcal{Q}_{subc} = \mathcal{Q}_{prod} = \{Mem, EQ\}$, since the algorithm is essentially the same if no equivalent queries are allowed. Unlike the other algorithms in this chapter, this algorithm assumes that an element from each hypothesis can be sampled. By "sample", we mean finding an arbitrary element that is in the hypothesis. There might not be an efficient algorithm for this sampling (e.g., if the concept class is NP-hard). However, this sampling will likely be more efficient than checking the equivalence of two concepts. Since the total number of samples from this algorithm is $k \cdot max_i\{\#EQ_i\}$, this sampling requirement is not too restrictive. The algorithm is shown in Algorithm 6 with a proof below.

Proposition 6. *If $\mathcal{Q}_{subc} = \mathcal{Q}_{prod} = \{\text{Mem}, \text{EQ}\}$, then Algorithm 6 can find a positive example using $max_i\{(\#\text{Mem}_i + \#\text{EQ}_i)\}^k$ membership queries and at most one equivalence query.*

Proof. Since the algorithm might give inconsistent answers to the sublearners, there is no guarantee that A_i can always give a new query. This is handled in the case marked "If no possible query" in the algorithm.

At the start of the algorithm, if the concept class includes the empty concept, the algorithm queries $\emptyset = c^*$. If $\emptyset = c^*$, the concept is learned. Otherwise, some positive counter-example from c^* must be given. The rest of the algorithm then assumes that is not a valid hypothesis.

The remaining part of the algorithm simulates each A_i in parallel until every T_i contains an element of c_i^*. At this point, the algorithm will stop, since all possible elements of $\prod T_i$ are posed as membership queries.

For each A_i, either every answer to a query from A_i is correct or at least one answer is incorrect. We will discuss each case separately. Every answer is correct: In this case A_i will eventually query the correct hypothesis c_i^*. Since $c_i^* \neq \emptyset$, some element of c_i^* will then be added to T_i. Some answer to A_i is incorrect: If an incorrect answer to a membership query is given, then for some query $x_i \in c_i^*$, the answer "False" is incorrect. So $x_i \in c_i^*$ and T_i will contain a positive example. If an incorrect answer to an equivalence query is given, either the counter-example is incorrect or the statement that they are not equivalent is incorrect. If an incorrect counterexample y_i to the query $S_i = c_i^*$ is given, then $y_i \in c_i^*$, since we already know $y_i \in S_i$. If the statement $S_i \neq c_i^*$ is incorrect, then some element of c_i^* will then be added to T_i.

The algorithm adds at most one element to T_i per query and must stop once every A_i has made enough queries to learn c_i^*, yielding the stated bound. □

Learning when only membership queries are allowed

As mentioned before, finding a positive example when $\mathcal{Q}_{subc} = \mathcal{Q}_{prod} = \{Mem\}$ is essentially the same as in Algorithm 6. The main difference is that we make the initial query $\emptyset = c^*$ at the beginning of the algorithm. For this reason, we need to assume that $\emptyset \notin C_i$ for all i. If not, there is no way to distinguish between an empty and non-empty concept. For example consider the classes

$C_1 = \{\{1\}, \emptyset\}$ and $C_2 = \{\{j\} \mid j \in \mathbb{N}\}$. It is easy to know when we have learned the correct class in C_1 or in C_2 using membership queries. However, learning from their cross-product is impossible. For any finite number of membership queries, there is no way to distinguish between the sets $\emptyset$ and $\{(1, j)\}$ for some j that has yet to be queried. By substituting in $\#EQ_i = 0$ for all i in the bound from Proposition 6, we get the following bound.

Proposition 7. *If $\mathcal{Q}_{subc} = \mathcal{Q}_{prod} = \{\text{Mem}\}$ and the empty set is not in the concept class, then the learner can find a positive example using at most $max_i\{\#\text{Mem}_i\}^k$ membership queries.*

Input: $\vec{p}$: Positive Example in X
Learn *($\vec{p}$)*
  while *some A_i has not completed* **do**
    for *each A_i* **do**
      // Answer *Mem* queries until an *EQ* query is made
      while *A_i queries $x_i \in C_i$ for some x_i* **do**
        Query $\vec{p}[i \leftarrow x_i] \in c^*$;
        Return answer to A_i;
      Receive query $S_i = c_i^*$ from A_i;
    Query $\prod S_i = c^*$;
    if $\prod S_i = c^*$ **then**
      return $\prod S_i$;
    else
      Receive counterexample $\vec{x} := (x_1, \ldots, x_k)$;
    if $\vec{x} \in \prod S_i$ **then**
      Set $i' := HandleCounterexample(\vec{p}, \vec{x})$;
      Pass $x_{i'}$ to $A_{i'}$ as a counterexample;
    else
      // $\vec{x} \in c^* \setminus \prod S_i$
      for $i \in \{1, \ldots, k\}$ **do**
        if $x_i \notin S_i$ **then**
          Pass x_i to A_i as a counterexample;
  return $\prod S_i$;

Algorithm 5: Learn when $\mathcal{Q}_{subc} = \{Mem, EQ\}$ and $\mathcal{Q}_{prod} = \{Mem, EQ, 1Pos\}$

4.6 Learning from Equivalence or Subset Queries is Hard

The previous section showed that learning cross products of membership queries requires at most $O(max_i\{\#Mem_i(c_i)\}^k)$ membership queries. A natural next question is whether this can be done for equivalence and subset queries. In this section, we answer that question in the negative. We will construct a class $\mathfrak{C}$ that can be learned from n equivalence or subset queries but which requires at least k^n queries to learn $\mathfrak{C}^k$. We define $\mathfrak{C}$ to be the set $\{\mathfrak{c}(s) \mid s \in \mathbb{N}^*\}$, where $\mathfrak{c}(s)$ is defined over strings such that $\mathfrak{c}(\lambda) := \{\lambda\} \times \mathbb{N}$, $\mathfrak{c}(s) := (\{s\} \times \mathbb{N}) \cup \mathfrak{c}_{sub}(s)$, and $\mathfrak{c}_{sub}(s \cdot a) := (\{s\} \times (\mathbb{N} \setminus \{a\})) \cup \mathfrak{c}_{sub}(s)$.

FindPos()
- **if** $\emptyset \in C$ **then**
  - Query $\emptyset = c^*$;
  - **return** counterexample;
- Initialize all $T_i := \emptyset$;
- **while** *True* **do**
  - **for** $i \in \{1, \ldots, k\}$ **do**
    - Ask A_i for query;
    - **if** *No possible query* **then**
      - Pass ;
    - **if** A_i *returns hypothesis* c_i **then**
      - Sample $y_i \in c_i$;
      - Add y_i to T_i;
    - **if** A_i *queries* $x_i \in c_i^*$ **then**
      - Add x_i to T_i;
      - Pass "False" to A_i;
    - **else**
      - A_i queries $S_i = c_i^*$;
      - Sample $y_i \in S_i$;
      - Pass y_i as counterexample to A_i;
      - Add y_i to T_i;
  - **for** *Unqueried* $\vec{y} \in \prod T_i$ **do**
    - Query $\vec{y} \in c^*$;
    - **if** $\vec{y} \in c^*$ **then**
      - **return** $\vec{y}$;

Algorithm 6: Finds positive example when $\mathcal{Q}_{subc} = \mathcal{Q}_{prod} = \{Mem, EQ\}$

For example, $\mathfrak{c}(1 \cdot 2) = (\{1 \cdot 2\} \times \mathbb{N}) \cup (\{1\} \times (\mathbb{N} \backslash \{2\})) \cup (\{\lambda\} \times (\mathbb{N} \backslash \{1\}))$. Here, $1 \cdot 2$ refers to the concatenation of symbols 1 and 2. To learn $\mathfrak{c}(s)$, it is enough to find the underlying string s. This can be done by constructing longer prefixes of s from the counter-examples given by an oracle.

For example, consider the tree in figure 4.5, where nodes represent potential hypotheses. Arrows are labelled with potential counterexamples, pointing to the next hypothesis to be considered given that counterexample. An algorithm learning $\mathfrak{c}(1 \cdot 2)$ from equivalence queries might start by querying $\mathfrak{c}(\lambda)$ and getting a negative counter-example $(\lambda, 1)$. It then queries $\mathfrak{c}(1)$ and gets a negative counter-example $(1, 2)$. Finally, it queries the correct concept $\mathfrak{c}(1 \cdot 2)$ and is done.

An important part of the construction of $\mathfrak{C}$ is that for any two strings $s, s' \in \mathbb{N}$, we have that $\mathfrak{c}(s) \subseteq \mathfrak{c}(s')$ if and only if $s = s'$. This implies that a subset query will return true if and only if the true concept has been found. Moreover, an adversarial oracle can always give a negative example for an equivalence query, meaning that oracle can give the same counterexample if a subset query were posed. So we will show that $\mathfrak{C}$ is learnable from equivalence queries, implying that it is learnable from subset queries.

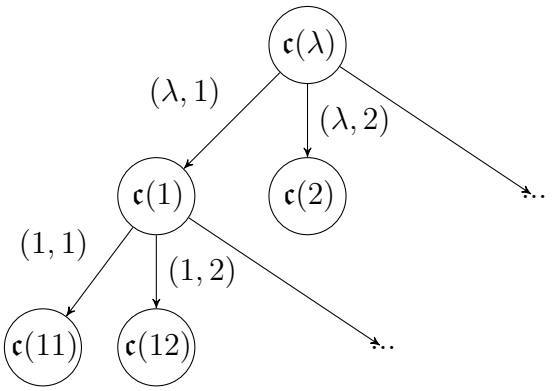

Figure 4.5: A tree representing Algorithm 7. Nodes are labelled with the queries made at each step, and edges are labelled with the counterexample given by the oracle.

Result: Learns $\mathfrak{C}$
Set $s = \lambda$;
while *True* **do**
  Query $\mathfrak{c}(s)$ to Oracle **if** *Oracle returns 'yes'* **then**
    return $\mathfrak{c}(s)$
  if *Oracle returns* $(s', m) \in c^* \backslash \mathfrak{c}(s)$ **then**
    Set $s = s'$;
  if *Oracle returns* $(s, m) \in \mathfrak{c}(s) \backslash c^*$ **then**
    Set $s = sm$;

Algorithm 7: Learning $\mathfrak{C}$ from equivalence queries.

We can conclude with this proposition.

Proposition 8. *There exist algorithms for learning from equivalence queries or subset queries such that any concept $\mathfrak{c}(s) \in \mathfrak{C}$ can be learned from $|s|$ queries.*

Proof. Algorithm 7 shows the learning algorithm for equivalence queries, and Figure 4.5 show the decision tree. This algorithm starts by querying $\mathfrak{c}(\lambda)$ to an oracle. When learning $\mathfrak{c}(s)$ for any $s \in \mathbb{N}^*$, the algorithm will construct s by learning at least one new element of s per query. Each new query to the oracle is constructed from a string that is a substring of s If a positive counterexample is given, this can only yield a longer substring of s and so learning is done in less than $|s|$ time. □

Showing $\mathfrak{C}^k$ is Hard to Learn

We will prove a lower-bound on learning $\mathfrak{C}^k$ from subset queries from an adversarial oracle. This will imply that $\mathfrak{C}^k$ is hard to learn from equivalence queries, since an adversarial equivalence query oracle can give the exact same answers and counterexamples as a subset query oracle.

It is easy to learn $\mathfrak{C}$, since each new counterexample gives one more element in the target string s. When learning a concept, $\prod \mathfrak{c}(s_i)$, it is not clear which dimension a given counterexample applies to. Specifically, a given counterexample $\vec{x}$ could have the property that $\vec{x}[i] \in \mathfrak{c}(s_i)$ for all $i \neq j$, but the learner cannot infer the value of this j. It must then proceed considering all possible values of j, requiring exponentially more queries for longer s_i.

To see this, consider the following example, where a learner must learn $\mathfrak{c}(s_1) \times \mathfrak{c}(s_2)$, when $|s_1| + |s_2| = 2$.

Example 10. *First, the learner queries* $(\mathfrak{c}(\lambda), \mathfrak{c}(\lambda))$ *to the oracle and receives a counter-example* $((\lambda, 1), (\lambda, 2))$. *It now knows either* s_1 *starts with* 1 *or* s_2 *starts with* 2. *The learner queries* $(\mathfrak{c}(1), \mathfrak{c}(\lambda))$ *and receives counterexample* $((1, 3), (\lambda, 4))$. *If* s_1 *starts with* 1*, then either* $s_1 = 1 \cdot 3$ *or* $s_2 = 4$. *If* s_1 *does not start with* 1*, then we've learned nothing. The learner queries* $(\mathfrak{c}(\lambda), \mathfrak{c}(2))$ *and receives counterexample* $((\lambda, 5), (2, 6))$. *At this point, there are four possible solutions:* $(\mathfrak{c}(1), \mathfrak{c}(4))$, $(\mathfrak{c}(1 \cdot 3), \mathfrak{c}(\lambda))$, $(\mathfrak{c}(5), \mathfrak{c}(2))$ *and* $(\mathfrak{c}(\lambda), \mathfrak{c}(2 \cdot 6))$. *The learner must query all but one of these in order to find the correct concept.*

A learning algorithm that uses the above strategy would need $2^{(|s_1|+|s_2|)}$ queries to learn $\mathfrak{c}(s_1) \times \mathfrak{c}(s_2)$.

Theorem 5. *Any algorithm learning* $\mathfrak{C}^k$ *from subset (or equivalence) queries requires at least* k^r *queries to learn a concept* $\prod \mathfrak{c}(s_i)$*, where* $r = \sum |s_i|$. *Equivalently, the algorithm takes* $k^{\sum \#q_i}$ *subset (or equivalence) queries.*

We will prove a lower-bound on learning $\mathfrak{C}^k$ from subset queries from an adversarial oracle. This will imply that $\mathfrak{C}^k$ is hard to learn from equivalence queries, since an adversarial equivalence query oracle can give the exact same answers and counterexamples as a subset query oracle.

First, we need a couple definitions.

A concept $\prod \mathfrak{c}(s_i)$ is *justifiable* if one of the following holds:

- For all i, $s_i = \lambda$
- There is an i and an $a \in \mathbb{N}$ and $w \in \mathbb{N}^*$ such that $s_i = wa$, and the k-ary cross-product $\mathfrak{c}(s_1) \times \cdots \times \mathfrak{c}(w) \times \cdots \times \mathfrak{c}(s_k)$ was justifiably queried to the oracle and received a counterexample $\vec{x}$ such that $\vec{x}[i] = (w, a)$.

A concept is *justifiably queried* if it was queried to the oracle when it was justifiable.

For any strings $s, s' \in \mathbb{N}^*$, we write $s \leq s'$ if s is a substring of s', and we write $s < s'$ if $s \leq s'$ and $s \neq s'$. We say that the *sum of string lengths* of a concept $\prod \mathfrak{c}(s_i)$ is of size r if $\sum |s_i| = r$

Proving that learning is hard in the worst-case can be thought of as a game between learner and oracle. The oracle can answer queries without first fixing the target concept. It will answer queries so that for any n, after less than k^n queries, there is a concept consistent with all given oracle answers that the learning algorithm will not have guessed. The specific behavior of the oracle is defined as follows:

- It will always answer the same query with the same counterexample.

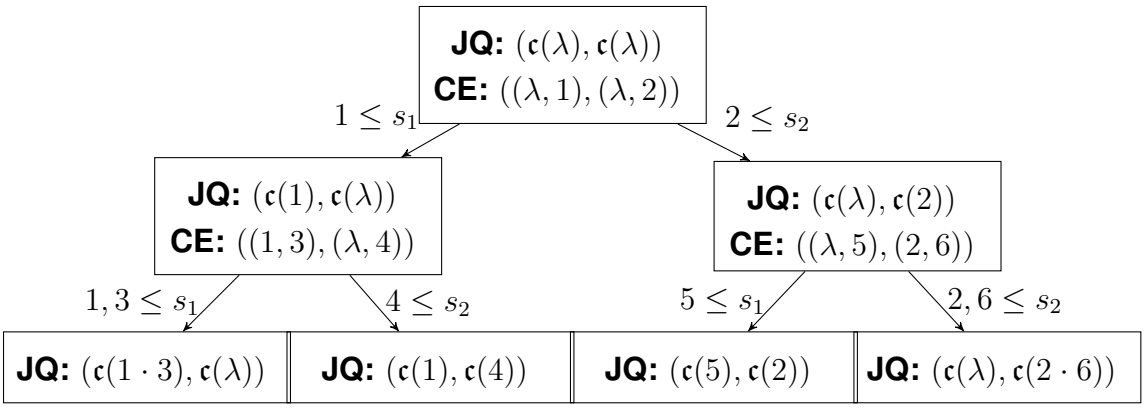

Figure 4.6: The tree of justifiable queries used in Example 11. Each node lists the justifiable query (JQ) and counterexample (CE) given for that query. The edges below each node are labelled with the possible inferences about s_1 and s_2 that can be drawn from the counterexample.

- Given any query $\prod \mathfrak{c}(s_i) \subseteq c^*$, the oracle will return a counterexample $\vec{x}$ such that for all i, $\vec{x}[i] = (s_i, a_i)$, and a_i has not been in any query or counterexample yet seen.
- The oracle never returns 'yes' on any query.

The remainder of this section assumes that queries are answered by the above oracle. An example of answers by the above oracle and the justifiable queries it yields is given below.

Example 11. *Consider the following example when $k = 2$. First, the learner queries $(\mathfrak{c}(\lambda), \mathfrak{c}(\lambda))$ to the oracle and receives a counter-example $((\lambda, 1), (\lambda, 2))$. The justifiable concepts are now $(\mathfrak{c}(1), \mathfrak{c}(\lambda))$ and $(\mathfrak{c}(\lambda), \mathfrak{c}(2))$. The learner queries $(\mathfrak{c}(1), \mathfrak{c}(\lambda))$ and receives counterexample $((1, 3), (\lambda, 4))$. The learner queries $(\mathfrak{c}(\lambda), \mathfrak{c}(2))$ and receives counterexample $((\lambda, 5), (2, 6))$. The justifiable concepts are now $(\mathfrak{c}(1), \mathfrak{c}(4))$, $(\mathfrak{c}(1 \cdot 3), \mathfrak{c}(\lambda))$, $(\mathfrak{c}(5), \mathfrak{c}(2))$ and $(\mathfrak{c}(\lambda), \mathfrak{c}(2 \cdot 6))$. At this point, these are the only possible solutions whose sum of string lengths is 2. The graph of justifiable queries is given in Figure 4.6.*

The following simple proposition can be proven by induction on sum of string lengths.

Proposition 9. *Let $\prod \mathfrak{c}(s_i)$ be a justifiable concept. Then for all w_1, w_2, ..., w_k where for all i, $w_i \leq s_i$, $\prod \mathfrak{c}(w_i)$ has been queried to the oracle.*

Proposition 10. *If all justified concepts $\prod \mathfrak{c}(s_i)$ with sum of string lengths equal to r have been queried, then there are k^{r+1} justified queries whose sum of string lengths equals $r + 1$*

Proof. This proof follows by induction on r. When $r = 0$, the concept $\prod \mathfrak{c}(\lambda)$ is justifiable.For induction, assume that there are k^r justifiable queries with sum of string lengths equal to r. By construction, the oracle will always chose counterexamples with as-yet unseen values in $\mathbb{N}$. So querying each concept $\prod \mathfrak{c}(s_i)$ will yield a counterexample $\vec{x}$ where for all i, $\vec{x}[i] = (s_i, a_i)$ for new

a_i. Then for all i, this query creates the justifiable concept $\prod \mathfrak{c}(s'_j)$, where $s'_j = s_j$ for all $j \neq i$ and $s'_i = \mathfrak{c}(s_i \cdot a_i)$. Thus there are k^{r+1} justifiable concepts with sum of string lengths equal to $r+1$. □

We are finally ready to prove the main theorem of this section.

Theorem Any algorithm learning $\mathfrak{C}^k$ from subset (or equivalence) queries requires at least k^r queries to learn a concept $\prod \mathfrak{c}(s_i)$, whose sum of string lengths is r. Equivalently, the algorithm takes $k^{\sum \#q_i}$ subset (or equivalence) queries.

Proof. Assume for contradiction that an algorithm can learn with less than k^r queries and let this algorithm converge on some concept $c = \prod \mathfrak{c}(s_i)$ after less than k^r queries. Since less than k^r queries were made to learn c, by Proposition 10, there must be some justifiable concept $c' = \prod \mathfrak{c}(s'_i)$ with sum of string lengths less than or equal to r that has not yet been queried. By Proposition 9, we can assume without loss of generality that for all $w_i \leq s'_i$, $\prod \mathfrak{c}(w_i)$ has been queried to the oracle. We will show that c' is consistent with all given oracle answers, contradicting the claim that c is the correct concept. Let $c_v := \prod \mathfrak{c}(v_i)$ be any concept queried to the oracle, and let $\vec{x}$ be the given counterexample. If for all i, $v_i \leq s'_i$, then by construction, there is a j with $\vec{x}[j] = (v_j, a_j)$ such that $v_j \cdot a_j \leq s'_j$, so $\vec{x}$ is a valid counterexample. Otherwise, there is an i such that $v_i \not\leq s'_i$. So $(\{v_i\} \times \mathbb{N}) \cap \mathfrak{c}(s'_i) = \emptyset$, so $\vec{x}$ is a valid counterexample. Therefore, all counterexamples are consistent with c' being correct concept, contradicting the claim that the learner has learned c. □

4.7 Efficient PAC-Learning

This sections discusses the problem of PAC-learning the cross products of concept classes.

Previously, [58] have shown the following bound on the VC-dimension of cross-products of sets:

$$\mathcal{VC}(\prod C_i) \leq a_1 log(ka_2) \sum \mathcal{VC}(C_i)$$

Here a_1 and a_2 are constants with $a_1 \approx 2.28$ and $a_2 \approx 3.92$. As always, k is the number of concept classes included in the cross-product.

The VC-dimension gives a bound on the number of labelled examples needed to PAC-learn a concept, but says nothing of the computational complexity of the learning process. This complexity mostly comes from the problem of finding a concept in a concept class that is consistent with a set of labelled examples. We will show that the complexity of learning cross-products of concept classes is a polynomial function of the complexity of learning from each individual concept class.

First, we will describe some necessary background information on PAC-learning.

PAC-learning Background

Definition 1. *Let C be a concept class over a space X. We say that C is efficiently PAC-learnable if there exists an algorithm A with the following property: For every distribution $\mathcal{D}$ on X, every $c \in C$, and every $\epsilon, \delta \in (0,1)$, if algorithm A is given access to $EX(c, \mathcal{D})$ then with probability $1-\delta$, A will return a $c' \in h$ such that $error(c') \leq \epsilon$. A must run in time polynomial in $1/\epsilon$, $1/\delta$, and $size(c)$.*

We will refer to ϵ as the 'accuracy' parameter and δ as the 'confidence' parameter. The value of $error(c)$ is the probability that for an x sampled from $\mathcal{D}$ that $c(x) \neq c^*(x)$. PAC-learners have a *sample complexity* function $m_C(\epsilon, \delta) : (0,1)^2 \rightarrow \mathbb{N}$. The sample complexity is the number of samples an algorithm must see in order to probably approximately learn a concept with parameters ϵ and δ.

Given a set S of labelled examples in X, we will use $A(S)$ to denote the the concept class the algorithm A returns after seeing set S .

A learner A is an *empirical risk minimizer* (ERM) if $A(C)$ returns a $c \in C$ that minimizes the number of misclassified examples (i.e., it minimizes $|\{(x,b) \in S \mid c^*(x) \neq b\}|$). The following theorem is well-known in PAC literature (Theorem 6.7 from [50])

Theorem 6. *If the concept class C has VC dimension d, then there is a constant, b, such that applying an Empirical Risk Minimizer A to $m_C(\epsilon, \delta)$ samples will PAC-learn in C, where*

$$m_C(\epsilon, \delta) \leq b\frac{d \cdot log(1/\epsilon) + log(1/\delta)}{\epsilon}$$

Essentially, this theorem is stating that an ERM can PAC learn a concept from polynomially many examples. Our goal is thus to show how to construct an ERM for the cross-products from the learning algorithms A_1 and A_2.

Finally, we will review the *growth function*, which describes how many distinct assignments a concept class can make to a given set of elements. More formally, for a concept class C and $m \in \mathbb{N}$, the growth function $G_C(m)$ is defined by:

$$G_C(m) = \max_{x_1, x_2, \ldots, x_m} \Big| \{(c(x_1), c(x_2), \ldots, c(x_m)) \mid c \in C\} \Big|$$

Each x_i in the above equation is taken over all possible elements of X_i. The VC-dimension of a class C is the largest number d such that $G_C(d) = 2^d$.

We will use the following bound, a corollary of the Perles-Sauer-Shelah Lemma, to bound the runtime of learning cross-products [50].

Lemma 4. *For any concept class C with VC-dimension d and $m > d + 1$:*

$$G_C(m) \leq (em/d)^d$$

PAC-Learning Cross-Products

We now have enough background to describe the strategy for PAC-learning cross-products. We will just describe learning the cross-product of two concepts. As above, assume concept classes C_1 and C_2 and PAC-learners A_1 and A_2 are given. We define $A_i(\epsilon, \delta)$ as the runtime of the sublearner A_i to PAC-learn with accuracy parameter ϵ and confidence parameter δ.

Assume that C_1 and C_2 have VC-dimension d_1 and d_2, respectively.

We can use the bound from van Der Vaart and Weller to get an upper bound d on the VC-dimension of their cross-product. Assume the algorithm is given an ϵ and δ and there is a fixed

target concept $c^* = c_1^* \times c_2^*$. Theorem 6 gives a bound on the sample complexity $m_{C_1 \times C_2}(\epsilon, \delta)$. The algorithm will take a sample of labelled examples of size $m_{C_1 \times C_2}(\epsilon, \delta)$. Our goal is to construct an Empirical Risk Minimizer for $C_1 \times C_2$. In our case, the target concepts c_1^* and c_2^* are in C_1 and C_2, respectively. Therefore, for any sample S, an Empirical Risk Minimizer will yield a concept in $C_1 \times C_2$ that is consistent with S. This process is shown in Algorithm 8.

We will now argue that Algorithm 8 is correct. Let S be any such sample the algorithm takes. This set can easily be split into positive examples S^+ and negative examples S^-, both in $X_1 \times X_2$. The algorithm works by maintaining sets labeled samples L_1 and L_2 for each dimension. For any $(x_1, x_2) \in S^+$, it holds that $x_1 \in c_1^*$ and $x_2 \in c_2^*$ so $(x_1, \top)$ and $(x_2, \top)$ are added to L_1 and L_2 respectively. For any $(x_1, x_2) \in S^-$, we know that $x_1 \notin c_1^*$ or $x_2 \notin c_2^*$ (or both), but it is not clear which is true. However, since the goal is only to create an Empirical Risk Minimizer, it is enough to find any concepts C_1 and C_2 that are consistent with these samples. Let $S_1^- := \{x \mid \exists y, (x, y) \in S^-\}$, let $m = |S_1^-|$ and order the elements of S_1^- by $x_1, x_2, \ldots, x_m$. The following lemma gives a bound on the number of concepts consistent with these examples.

Lemma 5. *Given the elements $x_1, \ldots, x_m$ in S_1^- as described above, the following bound holds $|\{(c(x_1), c(x_2), \ldots, c(x_m))| \leq (em/d)^d$.*

Proof. By the definition of growth function, $|\{(c(x_1), c(x_2), \ldots, c(x_m)) \mid c \in C_1\}| \leq G_{C_1}(m)$. By lemma 4, $G_{C_1}(m) \leq (em/d)^d$. □

In other words, there are less than $(em/d)^d$ assignments of truth values to elements of S_1^- that are consistent with some concept in C_1. If the algorithm can check every $c_1 \in C_1$ consistent with S^+ and S_1^-, it can then call A_2 to see if there is any $c_2 \in C_2$ such that $(c_1 \times c_2)$ assigns true to every element in S^+ and false to every element in S^-.

Finding these consistent elements of C_1 is made easier by the fact that we can check whether partial assignments to S_1^- are consistent with any concept in C_1. As mentioned above, it starts by creating the sets L_1 and L_2 containing all samples in the first and second dimension of S^+, respectively. It then iteratively adds labeled samples from S^-. At each step, the algorithm chooses one element $(x_1, x_2) \in S^-$ at a time and checks which possible assignments to x_1 are consistent with L_1. If $(x_1, \bot)$ is consistent, it adds $(x_1, \bot)$ to L_1 and calls $RecursiveFindSubconcepts$ on L_1 and L_2. If $(x_1, \top)$ is consistent with C_1, then the algorithm adds $(x_1, \top)$ to L_1 and $(x_2, \bot)$ to L_2 and calls RFS (RecursiveFindSubconcepts). In either case, if an assignment is not consistent, no recursive call is made. We can summarize these results in the following theorem.

Theorem 7. *Let concept classes C_1 and C_2 have VC-dimension d_1 and d_2, respectively. There exists a PAC-learner for $C_1 \times C_2$ that can learn any concept using a sample of size $m = ((d_1 + d_2) \cdot log(1/\epsilon) + log(1/\delta))/\epsilon$. The learner requires time $O(m^{d_1}(A_1(1/m, log(\delta)) + A_2(1/m, log(\delta))))$.*

Efficient PAC-learning with Membership Queries

Although polynomial, the complexity of PAC-learning cross-products from a *EX* oracle is fairly expensive. We will show that when a learner is allowed to make membership queries, PAC-learning

cross-products becomes much more efficient. This is due to the previously shown technique, which uses membership queries and a single positive example to determine on which dimensions a negatively labelled example fails.

In this case, assuming that $\emptyset \in \boxtimes C_i$, we can ignore the assumption that a positive example is given. If no positive example appears in a large enough labeled sample, the the algorithm can pose $\emptyset$ as the hypothesis.

If S does contain a positive example $\vec{p}$, then S can be broken down into labeled samples for each dimension i. The algorithm initialize the sets of positive and negative examples to $S_i^+ := \{\vec{x}[i] \mid (\vec{x}, \top) \in S\}$ and $S_i^- := \{\}$, respectively. For each $(\vec{x}, \bot) \in S$, a membership queries $\vec{p}[i \leftarrow \vec{x}[i]] \in c^*$. If so, $\vec{x}[i]$ is added to S_i^+. Otherwise it is added to S_i^-. This labelling is correct by Observation 2. The set of labelled examples $S_i := (S_i^+ \times \{\top\}) \cup (S_i^- \times \{\bot\})$ is then passed to the sublearner A_i. A_i is run on S_i with accuracy parameter $\epsilon' := \epsilon/k$ and confidence parameter $\delta' := \delta/k$.

Proposition 11. *The algorithm described above PAC-learns from the concept class $\boxtimes C_i$ with accuracy ϵ and confidence δ. It makes $m_C(\epsilon, \delta)$ queries to* EX*, $k \cdot m_C(\epsilon, \delta)$ membership queries, and has runtime $O(\sum A_i(\epsilon/k, \delta/k))$.*

Result: Find Subconcepts Consistent with Sample
Input: S^+: Set of positive examples in $X_1 \times X_2$
S^-: Set of negative examples in $X_1 \times X_2$
δ : Confidence parameter in $(0, 1)$
FindSubconcepts *(S^+, S^-, δ)*
    $\delta' := \delta/(|S^-|G_{C_1}(|S^-|) + G_{C_2}(|S^-|))$;
    $\epsilon' := 1/|S|$;
    // L_1: Labelled samples in X_1
    $L_1 := \{(x_1, \top) \mid \exists y, (x_1, y) \in S^+\}$;
    // L_2: Labelled samples in X_2
    $L_2 := \{(x_2, \top) \mid \exists y, (y, x_2) \in S^+\}$;
    $U := S^-$;
    return RFS(L_1, L_2, U, ϵ', δ') ;

Algorithm 8: PAC Learning

// Recursive Find Subconcepts
RFS *(L_1, L_2, U, ϵ', δ'):*
    if $U = \emptyset$ **then**
        if $A_2(L_2, \epsilon', \delta')$ **then**
            return $(A_1(L_1, \epsilon', \delta'), A_2(L_2, \epsilon', \delta'))$;
        else
            return $\perp$;
    Get $(x_1, x_2) \in U$;
    $U := U \backslash \{(x_1, x_2)\}$;
    // Attempts to label x_1 as false
    if $A_1(L_1 \cup \{(x_1, \perp)\}, \epsilon', \delta') \neq \perp$ **then**
        $L_1' := L_1 \cup \{(x_1, \perp)\}$;
        $c := RFS(L_1', L_2, U, \epsilon', \delta')$;
        if $c \neq \perp$ **then**
            return c;
    // Attempts to label x_1 as true
    if $A_1(L_1 \cup \{(x_1, \top)\}, \epsilon', \delta') \neq \perp$ **then**
        $L_1' := L_1 \cup \{(x_1, \top)\}$;
        $L_2' := L_2 \cup \{(x_2, \perp)\}$;
        $c := RFS(L_1', L_2', U, \epsilon', \delta')$;
        if $c \neq \perp$ **then**
            return c ;

Algorithm 9: Recursive helper function for the PAC-learning algorithm.

4.8 Conclusion

This chapter studies the problem of learning cross products of concept classes given algorithms for learning their components. We present tight bounds for several classes of queries, including membership, subset, superset and equivalence queries. We give a polynomial reduction algorithm for PAC-learning.

Part II

Syntax-Guided Synthesis

Chapter 5

Decidability of SyGuS Theories

5.1 Overview

In this chapter, we present a theoretical analysis of the syntax-guided synthesis problem. We analyze the decidability of the SyGuS problem for different classes of grammars and logics. For grammars, we consider arbitrary context-free grammars (used in the first SyGuS paper [4]) and tree grammars (used in the SyGuS competition [2]). For logics, we consider the major theories studied in satisfiability modulo theories (SMT) [10], including equality and uninterpreted functions (EUF), finite-precision bit-vectors (BV), and arrays – extensional or otherwise (AR), as well as theories with finite domains (FD). Our major results are as follows:

- For EUF, we show that the SyGuS problem is undecidable over tree grammars. These results extend straightforwardly for the theory of arrays. (See Section 5.2.)
- We present a decidable special case of the SyGuS problem for EUF, called *regular-EUF*, which is EXPTIME-complete. We prove that the sets of solutions to regular-EUF problems can be represented by regular tree languages, and vice versa. This represents the first SyGuS problem that is decidable, but not trivially decidable. (See Section 5.3.)
- For arbitrary theories with finite domains (FD) defined in Section 5.4, we show that the SyGuS problem is decidable for tree grammars.
- For BV, we show (perhaps surprisingly) that the SyGuS problem is undecidable for the classes of context-free grammars and tree grammars. (See Section 5.5.)

See Table 5.1 for a summary of our main results.

5.2 SyGuS-EUF is Undecidable

We use SyGuS-EUF to denote the class of SyGuS problems $(\varphi, \text{EUF}, G, f)$ where G is a grammar generating expressions that are syntactically well-formed expressions in EUF for f. In this section, we prove that SyGuS-EUF is undecidable. The proof of undecidability is a reduction from the simultaneous rigid E-unification problem (SREU) [23]. We say that a set $E := \{e_1, \ldots, e_l\}$ of equations between terms in $T(\Sigma, V)$ together with an equation e^* between terms in $T(\Sigma, V)$ forms

Theory \ Grammar Class	Regular Tree	Context-free
Finite-Domain	D	?
Bit-Vectors	U	U
Arrays	U	U
EUF	U	U
Regular-EUF	D	?

Table 5.1: Summary of main results, organized by background theories and classes of grammars. "U" denotes an undecidable SyGuS class, "D" denotes a decidable class, and "?" indicates that the decidability is currently unknown.

a *rigid expression*, denoted $E \vdash^r e^*$. A *solution* to $E \vdash^r e^*$ is a substitution σ, such that $e^*\sigma$ and $e_i\sigma$ are ground for each $e_i \in E$ and $E\sigma \vdash e^*\sigma$. Given a set S of rigid equations, the SREU problem is to find a substitution σ that is a solution to each rigid equation in S. It is known to be undecidable [23]. We start the reduction with constructing a boolean expression Φ_S for a given set of rigid equations S over alphabet Σ and variables $V := \{x_1, \ldots, x_m\}$. Let each $r_i \in S$ be $e_{i,1}, \ldots, e_{i,l_i} \vdash^r e_i^*$, where $e_{i,1}, \ldots, e_{i,l_i}$, and e_i^* are equations between terms in $T(\Sigma, V)$. We associate with each rigid expression $r_i \in S$ a boolean expression $\psi_i := \left(\bigwedge_{j=1,\ldots,l_i} e_{i,j}\sigma_f \wedge \bigwedge_{k\neq j} a_k \neq a_j\right) \rightarrow e_i{}^*\sigma_f$, where σ_f is the substitution $\{f(a_1)/x_1, \ldots, f(a_m)/x_m\}$. The symbol f is a unary function symbol to be synthesized and $a_1, \ldots, a_m$ are fresh constants ($a_i \notin \Sigma$ for all i). We set $\Phi_S := \bigwedge_i \psi_i$.

Next we give the grammar G_S, which generates the terms that may replace f in Φ_S. We define G_S to have the starting nonterminal A_1 and the following rules:

$$\begin{array}{ll} A_1 \rightarrow & ITE(x = a_1, S', A_2) \\ A_2 \rightarrow & ITE(x = a_2, S', A_3) \\ \ldots & \\ A_{m-1} \rightarrow & ITE(x = a_{m-1}, S', A_{m-1}) \\ A_m \rightarrow & ITE(x = a_m, S', \bot) \end{array}$$

where $\bot$ is a fresh constant ($\bot \notin \Sigma$ and $\bot \neq a_i$ for all i). Additionally, for each $g \in \Sigma$ we add a rule $S' \rightarrow g(S', \ldots, S')$, where the number of argument terms of g matches its arity.

Lemma 6. *The SREU problem S has a solution if and only if the SyGuS-EUF problem $\rho_S := (\Phi_S, EUF, G_S, f)$ has a solution over the ranked alphabet Σ.*

Proof. The main idea behind this proof is that each $f(a_i)$ in Φ_S represents the variable x_i in S. Any replacement to f found in G_S corresponds to a substitution on all variables x_i in S that grounds the equations in the SREU problem.

$\rightarrow$: Let $\sigma_u := \{u_1/x_1 \ldots, u_m/x_m\}$ be a solution to S, where each u_i is a ground term in $T(\Sigma)$. We consider the term $w(x) := ITE(x = a_1, u_1, ITE(x = a_2, u_2, \ldots, ITE(x = a_m, u_m, \bot)\ldots)$, which is in the language of the grammar G_S. To show that $\Phi_s\{w/f\}$ is valid, it suffices to show that for each model M of $\Sigma \cup \{a_1, \ldots, a_m\} \cup V$ and for each ψ_i we have $M \models \psi_i\{w/f\}$. If

$M \not\models [\bigwedge_{j=1,\ldots,l_i} e_{i,j}\sigma_f \wedge \bigwedge_{k\neq j} a_k \neq a_j]\{w/f\}$, then $M \models \psi_i\{w/f\}$ holds trivially. We handle the remaining case below, giving justifications to the right of each new equation.

1. Assume $M \models [\bigwedge_{j=1,\ldots,l_i} e_{i,j}\sigma_f \wedge \bigwedge_{k\neq j} a_k \neq a_j]\{w/f\}$
2. $M \models \bigwedge_{k\neq j} a_k \neq a_j$ (1)
3. For each j: $M \models w(a_j) = u_j$ (2)
4. For each j: $M \models (e_{i,j}\sigma_f)\{w/f\} \leftrightarrow e_{i,j}\sigma_u$ (3)
5. $M \models \bigwedge_{j=1,\ldots,l_i}(e_{i,j}\sigma_f)\{w/f\}$ (1)
6. $M \models \bigwedge_{j=1,\ldots,l_i} e_{i,j}\sigma_u$ (4, 5)
7. $\{e_{i,j} \mid j = 1,\ldots,m\}\sigma_u \vdash e_i^*\sigma_u$ (def. SREU)
8. $M \models e^*\sigma_u$ (6,7, Birkhoff's Thm.)
9. $M \models (e^*\sigma_f)\{w/f\}$ (3,8)

Therefore, $M \models \Phi_S$ and we get that w is a solution to the SyGuS problem ρ_S.

$\leftarrow$: Let $w(x)$ and σ_u be defined as before and assume that w is a solution to the SyGuS problem ρ_S. Each u_i in w is ground, since the nonterminal S' in G_S can only produce ground terms. Chose any $r_i \in S$. We will show for every model M on $\Sigma \cup V$, that if $M \models \bigwedge_{j=1,\ldots,l_i} e_{i,j}\sigma_u$ then $M \models e_i^*\sigma_u$. By Birkhoff's theorem, this implies $e_{i,1}\sigma_u, \ldots, e_{i,l_i}\sigma_u \vdash e_i^*\sigma_u$.

1. Assume $M \models \bigwedge_{j=1,\ldots,l_i} e_{i,j}\sigma_u$
2. Let $\hat{M}$ be a model over $\Sigma \cup V \cup \{a_1,\ldots,a_m\}$ such that $\hat{M} \restriction \Sigma \cup V = M$ and $\hat{M}$ assigns each a_i to a distinct new element not in $dom(M)$.
3. $\hat{M} \models w(a_j) = u_j$ (2)
4. For each j: $\hat{M} \models (e_{i,j}\sigma_f)\{w/f\} \leftrightarrow e_{i,j}\sigma_u$ (3)
5. $\hat{M} \models \bigwedge_{j=1,\ldots,l_i} e_{i,j}\sigma_u$ (1,2)
6. $\hat{M} \models \bigwedge_{j=1,\ldots,l_i}(e_{i,j}\sigma_f)\{w/f\}$ (4,5)
7. $\hat{M} \models \psi_i\{w/f\}$ (w is a SyGuS solution)
8. $\hat{M} \models (e_i^*\sigma_f)\{w/f\}$ (6, 7)
9. $\hat{M} \models e_i^*\sigma_u$ (3,8)
10. $M \models e_i^*\sigma_u$ (2,9)

Thus $e_{i,1}\sigma_u, \ldots, e_{i,l_i}\sigma_u \vdash e_i^*\sigma_u$ and σ_u is a solution to S. □

The main idea behind the proof of Lemma 6 is that each $f(a_i)$ in Φ_S represents the variable x_i in S. Any replacement to f found in G_S corresponds to a substitution on all variables x_i in S that grounds the equations in the SREU problem. Since the SREU problem is undecidable, this lemma immediately yields the following theorem.

Theorem 8. *The SyGuS-EUF problem is undecidable.*

A key step in the proof of Lemma 6 is the use of *ITE* statements to allow a single expression w to encode instantiations of multiple different variables. It remains open whether there exist alternative proofs of EUF undecidability that do not rely on *ITE* statements. However, we can show that SyGuS-EUF is undecidable when synthesizing two functions, even when no *ITE* operators are used.

To see this, consider a result from Veanes [61], which states that SREU is undecidable when only two variables are used. If f and f' have arity 0, then replacing them with some w and w' is equivalent to replacing first-order variables with ground terms. Given any rigid equation $r := e_1, \ldots, e_k \vdash^r e^*$ with variables x and y, we can create a formula $\psi_r := (\bigwedge_i e_i \rightarrow e^*)\{f/x, f'/y\}$. So, over a set S of rigid equations, we can create a formula $\phi := \bigwedge_{r_i \in S} \psi_{r_i}$. If f and f' can draw replacements from any term in $T(\Sigma)$, then it follows that S has a solution if and only if there are $w, w' \in T(\Sigma)$ such that $\phi\{w/f, w'/f'\}$ is valid.

We use SyGuS-$Arrays$ to denote the class of SyGuS problems $(\varphi, Arrays, G, f)$, where $Arrays$ is the theory of arrays [10], and G is a grammar such that $L(G)$ are syntactically well-formed expressions in $Arrays$ for f. There is a standard construction for representing uninterpreted functions as read-only arrays [10]. Therefore, the undecidability of SyGuS-$Arrays$ follows from the undecidability of SyGuS-EUF, as we state below.

Corollary 1. *The SyGuS-$Arrays$ problem is undecidable.*

5.3 Regular SyGuS-EUF

This section describes a decidable special case of the *SyGuS-EUF* problem, which we call *regular-EUF*.

Definition 2. *We call (ϕ, EUF, G, f) a* regular-EUF *problem if G is a regular tree grammar that contains no ITE expressions and ϕ is a* regular-EUF formula *as defined below.*

A regular-EUF *formula is a formula $\phi := \bigwedge_i \psi_i$ over some ranked alphabet Σ, where each ψ_i satisfies the following conditions:*

1. *It is a disjunction of equations.*
2. *It does not contain any ITE expressions.*
3. *It contains at most one occurrence of f per equation.*
4. *It satisfies one of the following cases:*
   - *Case 1: The symbol f only occurs in positive equations.*
   - *Case 2: The symbol f occurs in exactly one negative equation, and nowhere else.*

An equation is negative if it appears with a negation before it, and it is positive, otherwise. We define any disjunction ψ that satisfies the above conditions as regular. *We will refer to a regular ψ as case-1 or case-2, depending on which of the above cases is satisfied. Note that every regular-EUF formula is in conjunctive normal form.*

An interesting case of regular-EUF formulas are of the form: $\bigwedge_i e_i \rightarrow \bigwedge_j e'_j$, where each e_i is a positive equation containing no f symbols, and each e'_j is a (positive or negative) equation containing at most one occurrence of an f symbol. We can use the antecedent to specify the operators of an algebra and the consequent to specify a new function to be built using the operators.

Example 12. *Suppose we want to create a binary NAND operator using only the binary OR and unary NOT operators, we might use the regular-EUF formula:* $[OR(0,0) = 0 \wedge OR(0,1) = 1 \wedge OR(1,0) = 1 \wedge OR(1,1) = 1 \wedge NOT(1) = 0 \wedge NOT(0) = 1] \rightarrow [f(0,0) = 1 \wedge f(1,0) =$

$1 \wedge f(0,1) = 1 \wedge f(1,1) = 0]$. *The grammar of replacements to f(x,y) would then be given by:* $S \to NOT(S) \mid OR(S,S) \mid x \mid y$

To restrict the solution so that at most two NOT operators are used, we create non-terminals A_i *(for* $i = 0, 1, 2$*) such that at most* i *NOT operators can be produced from* A_i*:*

$$S \to A_2 \qquad A_2 \to NOT(A_1) \mid OR(A_2, A_0) \mid OR(A_0, A_2) \mid OR(A_1, A_1) \mid x \mid y$$
$$A_1 \to NOT(A_0) \mid OR(A_1, A_0) \mid OR(A_0, A_1) \mid x \mid y \qquad A_0 \to OR(A_0, A_0) \mid x \mid y$$

These grammars might yield $OR(NOT(x), NOT(y))$ *as a possible solution.*

We will show that for every regular ψ_i, we can construct a regular tree automaton A_{ψ_i} of polynomial size that accepts precisely the solutions to the SyGuS-EUF problem on ψ_i. The set of solutions to ϕ then becomes $L(G) \cap \bigcap_i L(A_{\psi_i})$, where G is the grammar of possible replacements. The grammar G can be represented as a deterministic bottom-up tree automaton A_G whose size is exponential in $|G|$ [18]. The product-automaton construction can be used to determine if $L(G) \cap \bigcap_i L(A_{\psi_i})$ is non-empty, which implies that a solution exists to the SyGuS problem. This gives us an algorithm to decide the regular-EUF problem in time $O(2^{|G|} \cdot \prod_i |\psi_i|)$.

The connection between sets of ground equations and regular tree languages was first observed by Kozen [40], who showed that a language L is regular if and only if there exist a set E of ground equations and collection S of ground terms such that $L = \bigcup_{s \in S} [s]_E$. We now prove the following (similar) theorem that states that a certain set of equivalence classes of a ground equational theory can be represented by a regular tree automaton.

Theorem 9. *Let* E *be a set of ground equations over the alphabet* Σ*, and let* C *be a subterm-closed set of terms such that every term in* E *is in* C*. There exists a regular tree automaton without accepting states* $A_{E,C} := (Q, \Sigma, \delta)$ *such that each state in* Q *represents an equivalence class of a term in* C*. That is, for all terms* $s, t \in T(\Sigma)$ *such that there exist terms* $s', t' \in C$ *so that* $s =_E s'$ *and* $t =_E t'$*, it holds that* $s =_E t$ *if and only if* $\delta^*(s) = \delta^*(t)$*.*

Proof. Let $Q := \{q_s \mid s \in C\}$. For each term $g(s_1, \ldots, s_k) \in C$, for $g \in \Sigma^{(k)}$, let $\delta(g, q_{s_1}, \ldots, q_{s_k}) = q_{g(s_1,\ldots,s_k)}$. We define the function $merge(q, q')$ to operate on $A_{E,C}$ as follows: First, add $\delta(g, q_1, \ldots, q_{i-1}, q, q_{i+1}, \ldots, q_k) = q''$ to δ for all $q_1, \ldots, q_{i-1}, q_{i+1}, \ldots, q_k, q''$ and g with $\delta(g, q_1, \ldots, q_{i-1}, q', q_{i+1}, \ldots, q_k) = q''$. Second, recursively call $merge(q'', q''')$ for all q'' and q''' with $\delta(g, q_1, \ldots, q_{i-1}, q', q_{i+1}, \ldots, q_k) = q''$ and $\delta(g, q_1, \ldots, q_{i-1}, q, q_{i+1}, \ldots, q_k) = q'''$. Finally, remove q' from Q.

We iteratively construct $A_{E,C}$ by calling $merge(q_s, q_t)$ for each $s = t$ in E. The following inductive argument shows that $A_{E,C}$ indeed represents the equivalence classes of terms in C.

We will first show after each call to merge that for any $s, t \in T(\Sigma)$, if $\delta(s) = \delta(t)$ then $s =_E t$. If no merges have been called then $\delta(s) = \delta(t)$ implies s equals t (and thus $s =_E t$). Now assume the property holds and $merge(q, q')$ is called for some $q, q' \in Q$. There are two cases. First, assume there is a $u = u' \in E$ such that $q = \delta(u)$ and $q' = \delta(u')$. For any s and t such that $\delta(s) = \delta(t) = q$ after the merge, either $\delta(s) = \delta(t)$ before the merge or without loss of generality $\delta(s) = q$ and $\delta(t) = q'$. By our assumption that the property held before the merge, $s =_E u$ and $u' =_E t$, so $s =_E t$ and the property still holds. Now assume that there is no such $u = u' \in E$. Then there is an

$f \in \Sigma^{(k)}$ and states $q_1, \ldots, q_k$ such that $\delta(f, q_1, \ldots, q_k) = q$ and q'. So if $\delta(s) = q$ and $\delta(t) = q'$, then there are $u_1, \ldots, u_k \in T(\Sigma)$ such that for each i, $\delta(u_i) = q_i$. So $s =_E f(u_1, \ldots, u_k) =_E t$.

For any two $s, t \in T(\Sigma)$ we will show by induction of the derivation from s to t in E that if $s =_E t$ then $\delta(s) = \delta(t)$. The base case, when $s = t$, trivially holds. Now assume $s \leftrightarrow_E u_1 \leftrightarrow_E \cdots \leftrightarrow_E u_r \leftrightarrow_E t$ and the property holds for ally derivations of length r. Since $s =_E u_r$, we know that $\delta(s) = \delta(u_r)$. Since $u_r \leftrightarrow_E t$, there is a context C and an equation $u = v \in E$ such that $C[l] = u_r$ and $C[r] = t$. Thus $\delta(l)$ must equal $\delta(r)$ after merging, so $\delta(C[l]) = \delta(C[r])$. Therefore, $\delta(s) = \delta(u_r) = \delta(t)$.

□

For any set of ground equations E and set of terms C, we will use $A_{E,C}$ to denote the automaton constructed from E and C as defined in the above proof.

Solving the SyGuS problem for $\psi = \bigwedge_i \psi_i$ is the problem of finding a w such that $\bigwedge_i \psi_i\{w/f\}$. Let $\psi_i := e_1 \vee e_2 \vee \ldots e_{k-1} \vee \neg e_k \vee \cdots \vee \neg e_{k+r}$ be a regular formula. We split equalities from inequalities and let $P := \{e_1, \ldots, e_{k-1}\}$ and $N := \{e_k, \ldots, e_{k+r}\}$. We can rewrite ψ to the normal form $(\bigwedge_{e \in N} e) \rightarrow (\bigvee_{e \in P} e)$. Then $\psi\{w/f\}$ is equivalent to $\bigvee_{e \in P} N\{w/f\} \vdash e\{w/f\}$.

The construction of the automaton A_{ψ_i} that represents the solutions to ψ_i depends on whether ψ_i is case-1 or case-2. We start with case-1 and choose some $s = t \in P$. Assume the symbol f does not occur in $s = t$. If $N \vdash s = t$, then ψ_i is trivially true. Otherwise, if $N \nvdash s = t$, then $s = t$ can be removed from ψ_i to yield an equally solvable formula. Now assume f occurs in $s = t$. Without loss of generality, f does not occur in t (by condition (3)) and there is a context B and a set of terms $s_1, \ldots, s_{arity(f)}$ such that $s = B[f(s_1, \ldots, s_{arity(f)})]$. Let $C := Subterms(N) \cup Subterms(\{s_1, \ldots, s_{arity(f)}\})$ and let $A_{N,C} := (Q, \Sigma, \delta)$. For each $q \in Q$, there is a ground term u_q such that $\delta^*(u_q) = q$. Let $Q' \subseteq Q$ be the set of states q such that $\delta^*(B[u_q]) = \delta^*(t)$. By Theorem 9, $N \vdash B[u_q] = t$ if and only if $q \in Q'$. Therefore, for any replacement, w, of f, $N \vdash (s = t)\{w/f\}$ if and only if $\delta^*(w(s_1, \ldots, s_{arity(f)})) \in Q'$.

Let $A_{s=t} := (Q, \Sigma \cup \{x_1, \ldots, x_{arity(f)}\}, \delta', Q')$ be a tree automaton with accepting states Q'. We here treat the x_i as constants and define, for each x_i, $\delta'(x_i) := \delta^*(s_i)$. And for each $u \in T(\Sigma)$, define $\delta'^*(u) := \delta^*(u)$. A simple inductive argument shows that this is a well-founded definition for δ' and that, for any replacement w of f, $\delta^*(w(s_1, \ldots, s_{arity(f)})) = \delta'^*(w(x_1, \ldots, x_{arity(f)}))$. Thus, $L(A_{s=t})$ defines the precise set of terms w such that $N \vdash (s = t)\{w/f\}$.

The set of solutions to ψ can be given by the automaton A_{ψ_i} whose language is $\bigcup_{s=t \in P} L(A_{s=t})$. This can be found in time exponential in $|N|$ using the product construction for tree automata [18].

Example 13. *Let $\psi := (g(a) = b \wedge g(b) = a) \rightarrow f(a) = g(g(b))$. Note that this is a case-1 regular-EUF clause. If we set $E := \{g(a) = b, g(b) = a\}$ and $C := \{a, b, g(a), g(b), g(g(b))\}$, then $A := A_{E,C}$ is the automaton from Figure 5.1 (excluding the accepting state marker and x transition). Since the argument of f in $f(a) = g(g(b))$ is a and A parses a to state-1, a transition from x to state-1 is added to A. Since $g(g(b))$ parses to state-2 in A, state-2 is set as an accepting state in A. So A accepts the replacements w to f such that $\psi\{w/f\}$ is valid.*

Assume ψ is case-2 and let $s = t$ be the equation in N that contains f. We choose some $u = u' \in P$ and show how to construct the automaton accepting the replacements w such that $N\{w/f\} \vdash u = u'$ holds.

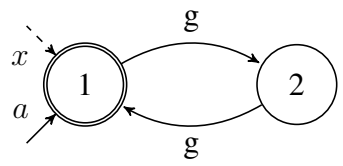

Figure 5.1: The automaton A_1 accepting the solutions to ψ_1 in example 13. This method of displaying tree automata is common in the literature [16]. For example, this automaton maps trees a, x, and $g(g(a))$ to state 1 and maps $g(a)$ and $g(x)$ to state 2.

Let $N' := N \backslash \{s = t\}$, and let $C := \mathit{Subterms}(N' \cup P) \cup \mathit{Subterms}(\{t, s_1, \ldots, s_{\mathit{arity}(f)}\})$. If $N' \vdash u = u'$ for some $u = u' \in P$, then every replacement to f is a solution.

So assume $N' \nvdash u = u'$. In this case we need the additional equality $s = t$ with a suitable replacement w for f to make $N\{w/f\} \vdash u = u'$ hold.

Without loss of generality, we assume that f does not occur in t (by condition (3)) and that there is a context B and a set of terms $s_1, \ldots, s_{\mathit{arity}(f)}$ such that $s = B[f(s_1, \ldots, s_{\mathit{arity}(f)})]$. Let $s' := s\{w/f\}$. We say that a term s is N'-equivalent to some term in C, if there exists a term $v \in C$ such that $s' =_{N'} v$.

We first show that to construct an automaton accepting the suitable replacements it suffices to consider the terms in C.

Lemma 7. *For every replacement w with $N\{w/f\} \vdash u = u'$ the term $w(s_1, \ldots, s_{\mathit{arity}(f)})$ is N'-equivalent to some term in C.*

Proof. Let $C' := C \cup \mathit{Subterms}(s')$ and let $A_{N',C'} = (Q, \Sigma, \delta)$. By way of contradiction assume that there is a replacement w such that $N\{w/f\} \vdash u = u'$ and $w(s_1, \ldots, s_{\mathit{arity}(f)})$ is not N'-equivalent to any term in C. Since C is subterm-closed, this means that s' is not N'-equivalent to any term in C and that s' is not a subterm of any term in C. In particular, s' is not a proper subterm of any term in C' and thus $\delta^*(s')$ has no outgoing edges in δ. By construction, $A_{N' \cup \{s'=t\},C'}$ is equivalent to calling $\mathit{merge}(\delta^*(s'), \delta^*(t))$ on $A_{N',C'}$. Since $\delta^*(s')$ has no outgoing edges, calling $\mathit{merge}(\delta^*(s'), \delta^*(t))$ on $A_{N',C'}$ cannot induce any more merges. Therefore, $\delta^*(u)$ and $\delta^*(u')$ are still not equal after the merge, contradicting the assumption $N\{w/f\} \vdash u = u'$. So s' and thus $w(s_1, \ldots, s_{\mathit{arity}(f)})$ is N'-equivalent to some term in C. □

We conclude the construction of the automaton in the following lemma.

Lemma 8. *The regular-EUF problem is in EXPTIME.*

Proof. As mentioned near the start of this section, to solve the regular-EUF problem, it suffices to construct an A_ψ that accepts precisely the replacements w that make ψ valid, for each ψ in ϕ. We have how to do this when ψ is case-1 and when ψ is case-2 and $N' \vdash u = u'$ for some $u = u' \in P$. We now handle the remaining case, when $N' \nvdash u = u'$.

Figure 5.2: Left: The set of solutions to ψ in Example 14. Right: The resulting automaton (without x transition and accepting state) after merging states 1 and 3.

Let $A_{N',C} := (Q, \Sigma, \delta)$. For each $q \in Q$, there is a ground term u_q such that $\delta^*(u_q) = q$. Let $Q_{u=u'}$ be the set of states such that $N' \cup \{B[u_{q'}] = t\} \vdash u = u'$ for each $q' \in Q_{u=u'}$. Then by Lemma 7, for each replacement w, $N\{w/f\} \vdash u = u'$ if and only if $\delta^*(w) \in Q_{u=u'}$.

Let $Q' := \bigcup_{e \in P} Q_e$. Let $A_\psi := (Q, \Sigma \cup \{x_1, \ldots, x_{arity(f)}\}, \delta', Q')$ be a tree automaton with accepting states Q'. For each x_i, let $\delta'(x_i) := \delta(s_i)$. For all $u \in T(\Sigma)$, let $\delta'(u) := \delta(u)$. A simple inductive argument will show that $L(A_\psi)$ contains precisely the solutions to ψ. □

Example 14. *Let $\psi := (g(h(g(a))) = a \wedge f(g(a)) = a) \rightarrow h(g(a)) = a$. Note that this is a case-2 regular-EUF clause. If we set $E := \{g(h(g(a))) = a\}$ and $C := \{a, g(a), h(g(a)), g(h(g(a)))\}$, then $A := A_{E,C}$ is the automaton from the left side of Figure 5.2 (excluding the accepting state and x transition). Since the argument of f in $f(g(a)) = a$ is a and A parses a to state-2, a transition from x to state-2 is added to A. If we choose a replacement w such that $w(g(a))$ parses to state-3 in A, then applying the equation $w(g(a)) = a$ merges state-3 with state-1. This, in turn, forces a merge between the new state and state-2, yielding the automaton on the right side of Figure-5.2. This automaton parses $h(g(a))$ and a to the same state, so state-3 is an accepting state. This does not occur if $w(g(a))$ parses to state-1 or state-2 in A, so they are not accepting states. So A accepts the replacements w to f such that $\psi\{w/f\}$ is valid.*

Lemma 9. *Let $A := (Q, \Sigma \cup \{x_1, \ldots, x_k\}, \delta, Q_F)$ be a tree automaton. There exists a regular disjunctive formula ψ_A such that $L(A)$ is the set of solutions to ψ_A.*

Proof. Let T_Q be a subterm-closed set of terms such that for each state $q \in Q$, there is a term u_q such that $\delta(u_q) = q$. Without loss of generality, assume that each $u_q \in T_Q$ is a subterm of some term in $L(A)$. Let $\sigma := \{x_i/c_i \mid i \in \{0, \ldots, k\}\}$ for some new constants $c_1, \ldots, c_k$. Let $N_Q := \{g(u_{q_1}\sigma, \ldots, u_{q_r}\sigma) = u_{q'}\sigma \mid r \geq 0, g \in \Sigma_r, q_1, \ldots, q_r, q' \in Q, \delta(g, q_1, \ldots, q_r) = q'\}$ and $P_Q := \{f(c_1, \ldots, c_k) = u_q \mid q \in Q_F\}$. Finally set $\psi_A := (\bigwedge_{e \in N_Q} e) \rightarrow (\bigvee_{e \in P_Q} e)$. Using the construction from theorem 9, it is easy to check that the set of solutions to ψ are precisely $L(A)$. □

We can also use the above lemma to show that regular SyGuS-EUF is EXPTIME-complete.

Lemma 10. *The regular-EUF problem is EXPTIME-hard.*

Proof. We reduce from the EXPTIME-complete problem of determining whether a set of regular tree automata have languages with a non-empty intersection [60]. Let $A_1, \ldots, A_k$ be a set of regular tree automata over some alphabet Σ. For each automaton A_i, construct the formula ψ_{A_i} as described in Lemma 9. Let $\phi := \bigwedge_i \psi_{A_i}$. Let f be a nullary function symbol to be synthesized, and let G be a grammar such that $L(G) := T(\Sigma)$. The solutions to the regular SyGuS-EUF problem (ϕ, Σ, G, f) are the members of the set $\bigcap_i L(A_i)$. Therefore, (ϕ, Σ, G, f) has a solution if and only if $\bigcap_i L(A_i)$ is non-empty. □

Lemma 8 and Lemma 10 give us the main theorem of Section 5.3.

Theorem 10. *The regular-EUF problem is EXPTIME-complete in the size of the grammar and the specification.*

Notice that if the number of occurrences of f is bounded and the grammar allows for all well-formed replacements, then the regular-EUF problem becomes polynomial. Additionally, the constructions above provide a novel characterization of regular tree languages.

Theorem 11. *A tree language is regular if and only if it is the set of solutions to a regular-EUF problem.*

Note that the construction from Lemma 9 does not require the use of a grammar of possible solutions. Therefore the regular-EUF formulae themselves provide a means of representing regular tree languages.

It is worth noting that the case-1 and case-2 restrictions in the definition of regular-EUF are necessary for the above theorem to hold. Specifically, if the definition of regular-EUF formulas were altered so that an f-symbol could appear in one positive and one negative equation (violating case-1) or two negative equations (violating case-2), then the set of possible solutions might not be a regular tree language.

Case-1 and Case-2 are necessary

While Theorem 11 already suggests that this decidable case of SyGuS cannot be easily generalized, we want to demonstrate along an example that restrictions in case-1 and case-2 are necessary. The following example gives a clause that includes one positive and one negative equation in which an f appears. The set of solutions to the corresponding SyGuS-EUF problem is not a regular tree language. More specifically:

Let $\Sigma := \{g:1, g':1, h:1, a:0, b:0, c:0\}$ be a ranked alphabet, and let f be a unary function symbol to be synthesized. Let $N := \{f(a) = b, g(a) = a, g'(a) = a, h(a) = b, h(b) = c, g(c) = c, g'(c) = c\}$ and $\phi := (\bigwedge_e \in Ne) \rightarrow f(b) = c$ (so $f(b) = c$ is positive and $f(a) = b$ is negative in the disjunctive form of ϕ). Define G to be the tree grammar with start symbol S and the following rules: $S \rightarrow g(S)|g'(S)|h(A)$ and $A \rightarrow g(A)|g'(A)|x$. We will show that the set of solutions to the regular-EUF problem (ϕ, Σ, G, f) is not a regular tree language.

Let $w(x) \in L(G)$ be a replacement to f and $E' := E\{w/f\}$. By the rules of G, there must be a context B and a term $t(x)$ over the alphabet $\{g:1, g':1\}$ such that $w(x) = B[h(t(x))]$. We can

see that $b =_{E'} w(a) =_{E'} B[h(t(a))] =_{E'} B[h(a)] =_{E'} B[b]$. Also, $w(b) =_{E'} c \Leftrightarrow h(t(b)) =_{E'} c \Leftrightarrow t(b) =_{E'} b$. The terms $t(x)$ such that $t(b) = b$ are precisely those of the form $B[B[\dots B[x]\dots]]$. Therefore, the set of solutions to the above regular-EUF is $L := \{B[h(B[B[\dots B[x]\dots]])] \mid B \text{ is any context over } \{g{:}1, g'{:}1\}\}$.

We now use the Myhill-Nerode theorem for regular tree languages [41], stated below:

Theorem 12 (Myhill-Nerode theorem for regular tree languages [18])**.** *Given a tree language L over ranked alphabet Σ, we define $s \equiv_L t$ if $C[s] \Leftrightarrow C[t]$ for each context C and terms s and t over Σ. The following are equivalent:*

1. *L is regular*
2. *$\equiv_L$ has finitely many equivalence classes*
3. *L is accepted by a rational tree automaton.*

Using this theorem, it is easy to check that L is not regular.

We can use this same example to show that the requirement in case-2 that f appears in at most one negative equation is also necessary. Let $N' := N \cup \{f(b) = d\}$. It is easy to check that solutions to the constraint $\phi' := (\bigwedge_e \in Ne) \rightarrow d{=}c$ are precisely the solutions to ϕ as defined above. Since N' contains both $f(b) = d$ and $f(a) = b$, we can see that the requirement in case-2 is necessary.

5.4 Finite-Domain Theories

In addition to the "standard" theories, we also consider a family of theories that we term *finite-domain (FD) theories*. Formally, an FD theory is a complete theory that admits one domain (up to isomorphism), and whose only domain is finite. For example, consider group axioms with a constant a and the statements $\forall x : (x = 0) \vee (x = a) \vee (x = a \cdot a)$ and $a \cdot a \cdot a = 0$. This is an FD theory, since, up to isomorphism, the only model of this theory is the integers with addition modulo 3. Also Boolean logic and the theory of fixed-length bit-vectors without concatenation are FD theories. Bit-vector theories with (unrestricted) concatenation allow us to construct arbitrarily many distinct constants and are thus not FD theories.

In this section we give a generic algorithm for any complete finite-domain theory for which validity is decidable. Let $\mathcal{T}$ be a such a theory and let M be a model of $\mathcal{T}$ with a finite domain *dom*(M). Assume without loss of generality that for every element $c \in$ *dom*(M) there is a constant f in M such that $f^M = c$.

We consider a SyGuS problem with a correctness specification φ in theory $\mathcal{T}$, a function symbol f to synthesize, and a tree grammar $G = (N, S, \mathcal{F}, P)$ generating the set of candidate expressions. Let $\mathbf{a} := a_1, \dots, a_r$ be the constants occurring in φ. The expression e generated by G to replace f can be seen as a function mapping $\mathbf{a}$ to an element in *dom*(M). If the domain of M is finite there are only finitely many candidate functions, but it can be non-trivial to determine which functions can be generated by G. In the following, we describe an algorithm that iteratively determines the set of functions that can be generated by each non-terminal in the grammar G.

For each $V \in N$, we maintain a set E_V of expressions e. In each iteration and for each production rule $V \rightarrow f(t_1, \dots, t_k)$ for V in G, we consider the expressions $f(t_1^*, \dots, t_k^*)$ where

Iteration#	E_S	E_A	E_B
1	none	x	y
2	$x \oplus y$	$\neg y$	none
3	$\neg y \oplus y \equiv \top$	none	$(x \oplus y) \oplus \neg y \equiv \neg x$ ~~$(x \oplus y) \oplus x \equiv y$~~
4	$\neg y \oplus \neg x$ ~~$\neg x \oplus x \equiv \top$~~	~~$\neg\neg x \equiv x$~~ none	~~$\top \oplus \neg y \equiv y$~~ ~~$\top \oplus x \equiv \neg x$~~ none
5	none	none	~~$(\neg y \oplus \neg x) \oplus \neg y \equiv \neg x$~~ ~~$(\neg y \oplus \neg x) \oplus x \equiv y$~~ none

Table 5.2: This table shows the expressions added to the sets E_S, E_A, and E_B when we apply the algorithm to the SyGuS problem in Example 15. For readability, we simplify the expressions, indicated by the symbol '$\equiv$'. Expressions that are syntactically new, but do not represent a new function are struck out. When no new function is added, "none" is written in the cell.

$t_i^* := t_i$ if t_i is an expression (i.e. $t_i \in T_{\mathcal{F}}$) and $t_i^* \in E_{V'}$ if t_i is a non-terminal V'. Given such an expression e, we compute the function table, that is the result of $e\{\mathbf{c}/\mathbf{a}\}$ for each $\mathbf{c} \in \mathit{dom}(M)^r$, compare it to the function table of the expressions currently in E_V. Our assumption of decidability of the validity problem for $\mathcal{T}$ guarantees that this operation is decidable. If e represents a new function, we add it to the set E_V.

The algorithm terminates, after an iteration in which no set E_V changed. As there are only finitely many functions from $\mathit{dom}(M)^r$ to $\mathit{dom}(M)$ and the sets E_V grow monotonously, the algorithm eventually terminates. To determine the answer to the SyGuS problem, we then check whether there is an expression e in E_S, for which $\varphi\{e/f\}$ is valid.

Theorem 13. *Let $\mathcal{T}$ be a complete theory for which validity is decidable and which has a finite-domain model M. The SyGuS problem for $\mathcal{T}$ and $\mathcal{T}$-compatible tree grammars is decidable.*

Example 15. *Consider the SyGuS problem over boolean expressions with the specification $\varphi = x \oplus f$, where $\oplus$ denotes the XOR operation and f is the function symbol to synthesize from the following tree grammar (we use infix operators for readability):*

$$S \to (A \oplus B) \qquad A \to \neg B \mid x \qquad B \to (S \oplus A) \mid y$$

The grammar generates boolean functions of variables x and y and the updates to E_A, E_B, and E_S during each iteration of the proposed algorithm are given in Table 5.2. The next step in the algorithm is to determine if any of the three expressions $E_S := \{x \oplus y, \neg y \oplus y, \neg y \oplus \neg x\}$ make the formula $\varphi\{e/f\}$ valid, which is not the case.

5.5 Bit-Vectors

In this section, we show that the SyGuS problem for the theory of bit-vectors is undecidable even when we restrict the problem to tree grammars. The proof makes use of the fact that we can construct (bit-)strings with the concatenation operation and can compare arbitrarily large strings with the equality operation. This enables encoding the problem of determining if the languages of CFGs with no ε-transitions have non-empty intersection, which is undecidable [31].

Theorem 14. *The SyGuS problem for the theory of bit-vectors is undecidable for both the class of context-free grammars and the class of BV-compatible tree grammars.*

Proof. We start with the proof for the class of context-free grammars. Given two context-free grammars $G_1 = (N_1, S_1, T_1, R_1)$ and $G_2 = (N_1, S_2, T_2, R_2)$, we define a SyGuS problem with a single context-free grammar $G = (N, S, T, R)$ that has a solution iff the intersection of G_1 and G_2 is not empty. The proof idea is to express the intersection of the two grammars as the equality between two expressions, each generated by one of the grammars. The new grammar thus starts with the following production rule: $S \to S_1 = S_2$.

We then have to translate the grammars G_1 and G_2 into grammars G_1' and G_2' that produce expressions in the bit vector theory instead of arbitrary strings over their alphabets. There is a string produced by both G_1 and G_2 if and only if the constructed grammars G_1' and G_2' can produce a pair of equal expressions. We achieve this by encoding each letter as a bit string of the fixed length $1 + \log_2 |T_1 \cup T_2|$, and by intercalating concatenation operators ($@$) in the production rules: We encode each production rule (N, P) with $P = p_1 p_2 \dots p_n$ as (N, P') with $P' = p_1' @ p_2' @ \dots @ p_n'$, where $p_i' = p_i$ if $p_i \in N$, and otherwise p_i' are the fixed-length encodings of the terminal symbols. We then define $N = S \dot\cup N_1' \dot\cup N_2'$, $T = \{0, 1, @, =\}$, and $R = R_1' \cup R_2' \cup \{(S, S_1' = S_2')\}$. The correctness constraint φ of our SyGuS problem then only states $\varphi \coloneqq \neg f$, where $f : \mathbb{B}$ is the function symbol, a constant, to synthesize. As each character in the alphabets of the context-free grammars was encoded using bit vectors of the same length, the comparison of bit vectors is equivalent to the comparison between strings of characters of the grammars G_1 and G_2 and the SyGuS problem has a solution if and only if the intersection of the languages of the context-free grammars G_1 and G_2 is empty.

Note that the context-free grammar G can also be interpreted as a BV-compatible tree grammar, where BV is the theory of bit-vectors. Although it is efficiently decidable whether two tree grammars produce a common tree, the expressions produced by the tree-interpretation of G_1 and G_2 will be equivalent as long as their leaves are equivalent. Thus, the equality of the expression trees in the interpretation of the bit vector theory still coincides with the intersection of the given context-free grammars G_1 and G_2. □

We only used the concatenation operation of the bit-vector theory for the proof. That is, SyGuS is even undecidable for fragments of the theory of bit-vectors for which basic decision problems are easier than the general class; for example, the theory of *fixed-sized bit-vectors with extraction and composition* [21] for which satisfiability of conjunctions of atomic constraints is polynomial-time solvable unlike the general case which is NP-hard.

Remark 1. *This proof only relies on the comparison of arbitrarily large values in the underlying logical theory. It may thus be possible to extend the proof to other theories involving numbers, such as LIA, LRA, and difference logic. The problem here is that these proofs tend to depend on syntactical sugar. Consider the case of LIA. If the signature allows us to use arbitrary integer constants, such as 42, it is simple to translate the proof above into a proof of undecidability of SyGuS for LIA and CFGs. For the standard signature of LIA, however, which just includes the integer constants 0 and 1 (larger integers can then be expressed as the repeated addition of the constant 1) the proof scheme above does not apply.*

5.6 Other Background Theories

In this section, we remark on the decidability for some other classes of SyGuS problems. These results are straightforward, but the classes occur in practice, and so the results are worth mentioning.

Linear real arithmetic (LRA) with arbitrary affine expressions.

Consider the family of SyGuS problems where:

i) the specification φ is a Boolean combination of linear constraints over real-valued variables $\vec{x} := x_1, x_2, \ldots, x_n$ and applications of the function f to be synthesized. For simplicity, we assume a single function f of arity n; the arguments below generalize.

ii) The grammar G is the one generating *arbitrary affine expressions* over $\vec{x}$ to replace for applications of f. Thus, the application $f(\vec{t})$, where $\vec{t} := t_1, t_2, \ldots, t_n$ is a vector of LRA terms, is replaced by an expression of the form $a_0 + \sum_{i=1}^{n} a_i t_i$.

Thus, for a fixed set of variables $\vec{x}$ there is a fixed grammar for all formulas φ.

This case commonly arises in invariant synthesis when the invariant is hypothesized to be an affine constraint over terms in a program. In this case, the solution of the SyGuS problem reduces to solving the $\exists\forall$ SMT problem

$$\exists a_0, a_1, \ldots, a_n \,.\, \forall x_1 x_2 \ldots x_n \,.\, \left(\varphi[f(\vec{t})/a_0 + \sum_{i=1}^{n} a_i t_i]\right)$$

which reduces to a formula with first-order quantification over real variables. Since the theory of linear real arithmetic admits quantifier elimination, the problem is solvable using any of a number of quantifier elimination techniques, including classic methods such as Fourier-Motzkin elimination [22] and the method of Ferrante and Rackhoff [27], as well as more recent methods for solving exists-forall SMT problems (e.g., [26]).

This decidability result continues to hold for grammars that generate bounded-depth conditional affine expressions. However, the case of unbounded-depth conditional affine expressions is still open, to our knowledge.

A similar reduction, for the case of affine expressions, can be performed for linear arithmetic over the integers (LIA), requiring quantifier elimination for Presburger arithmetic. Thus, this case is also decidable.

Finite-precision bit-vector arithmetic (BV) with arbitrary bit-vector functions.

Consider the family of SyGuS problems where:

i) the specification φ is an arbitrary formula in the quantifier-free theory of finite-precision bit-vector arithmetic [9] over a collection of k bit-vector variables whose cumulative bit-width is w. Let f be a bit-vector function to be synthesized with output bit-width m.

ii) The grammar G is the one generating *arbitrary bit-vector expressions* over these k variables, using all the operators defined in the theory. In other words, G imposes no major syntactic restriction on the form of the bit-vector function f.

Thus, for a fixed set of bit-vector variables there is a fixed grammar for all formulas φ.

This class of SyGuS problems has been studied as the synthesis of "bitvector programs" (in applications such as code optimization and program deobfuscation) from components (bit-vector operators and constants) [34]. It is easy to see that this class is decidable. A simplistic (but inefficient) way to solve it is to enumerate all 2^{m2^w} possible semantically-distinct bitvector functions over the k variables and check, via an SMT query, whether each, when substituted for f will make the resulting formula valid.

5.7 Discussion

The main results of this section are summarized in Table 5.1. We conclude with a few remarks about the results, connections between them, and their relevance in theory and practice.

Consider the theory of finite-precision bit-vector arithmetic (BV). We have seen that the SyGuS problem is undecidable when an arbitrary context-free grammar can be used to restrict the space of bit-vector functions to be synthesized (see Section 5.5). However, it is not hard to see that the SyGuS problem is decidable when the logical formula is an arbitrary BV formula and the grammar allows the function to be replaced by *any* bit-vector function over the constants in the formula.These results may seem to contradict the intuition that restricting the search space for synthesis with a grammar makes the problem easier to solve. We thus have to be careful which classes of grammars we pick to restrict SyGuS problems.

Practical approaches to SyGuS often restrict the grammar to admit only a finite number of expressions, e.g., by describing the expression with a finite number of bounded, discrete parameters [53], by restricting the number of times each syntactic element may be used [34] or by bounding the size of the expressions [57]. In this case the SyGuS problem is trivially decidable as long as the underlying SMT theory is decidable. With the connection to regular tree automata in Section 5.3 and the general algorithm for finite-domain theories in Section 5.4, we present the first non-trivial, decidable SyGuS problems.

For future work, it would be interesting to study the linear integer and real arithmetic background theories in more detail. In particular, we would like to determine if these theories are decidable when grammars are provided, and whether the use of conditionals without bounding expression tree depth affects the decidability. Further, for SyGuS classes that are decidable, it would be useful to perform a more fine-grained characterization of problem complexity, especially with regard to special classes of grammars.

Chapter 6

SyGuS with Costs

One of the most successful methods for SyGuS solving uses enumeration [4, 1]. An enumerative SyGuS solver enumerates all of the possible programs given by the grammar until it obtains one which satisfies the specification. New candidates are checked against counterexamples from previous calls to the verification oracle to save unnecessary work: if a candidate fails on an existing counterexample, we can immediately discard it.

A primary factor in the efficiency, and quality of output, of enumerative learning is the number of calls to the verification oracle required, which is in part determined by the order in which the enumeration algorithm explores the program space.

The ordering usually considered is *program length*, roughly the number of tokens in the program itself. Hence if synthesis succeeds, we obtain a correct program of minimal length [3, 56].

This can be described via a *cost metric* on programs v; the learning algorithm guarantees that, for any programs P_1, P_2, if $v(P_1) < v(P_2)$ then program P_1 is enumerated before program P_2. This in turn guarantees that when we find a program P which satisfies the specification, it is a minimal such program with respect to v.

The aim of this chapter is to provide an efficient method for enumerative learning with respect to other complexity metrics, which may be more suitable measures of program quality than the length. For example, perhaps some atomic operations are more costly than others, or operations compose in ways which are not represented by program length.

Our contributions

We employ the "k-best parsing" algorithm of [15] in the context of enumerative learning. We recall that this is a dynamic programming based algorithm for finding the k best derivations of a regular tree grammar (RTG) with "superior" cost functions. This generalises an algorithm of Knuth [39] for finding the shortest derivation.

We observe that this algorithm in fact applies to a wider class of cost functions called "weakly superior" [46]. This includes important cost functions which are not superior, such as the minimum and constant functions. In more detail, we obtain the following theorem; a weakly superior RTG is a RTG whose associated cost functions are all weakly superior.

Theorem 15. *Let G be a weakly superior regular tree grammar with M productions, N nonterminals and maximum production arity R. Then we can obtain a data structure, from which the k shortest derivations from G may be obtained in linear time, in time $\tilde{O}(M + kNR)$.*

Related work

Hu and D'Antoni [32] consider the task of synthesising programs with quantitative objectives using the formalism of weighted grammars. Their approach, QSyGuS, reduces solving SyGuS with quantitative constraints over a weighted grammar to solving (standard) SyGuS over a related unweighted grammar. Minimising a quantitative objective is achieved by repeatedly strengthening constraints until the problem becomes infeasible. The cost functions supported by QSyGuS are those which can be expressed by weighted grammars; in particular the cost of a derivation is the product of the costs of the productions it contains. We believe that our set of supported cost functions is incomparable to this.

Bornholt et al. [12] devise a general framework for optimal synthesis called *metasketches*. A metasketch is an ordered set of sketches (programs with holes [52]) with an associated cost function and 'gradient function'. The gradient function, on input a cost bound c, outputs a set of sketches S such that every program of cost at most c can be obtained from a sketch in S. The framework supports generic cost functions subject to some finiteness constraint, although the precise implementation of the gradient function is not discussed, and the cost functions used in the examples are supported by our algorithm.

6.1 Preliminaries

Grammars and Derivation Trees

Recall the definition of regular tree grammars given in the background section of this thesis, where a grammar G is a tuple (N, S, Σ, R) defined over a ranked alphabet Σ.

For a grammar G with nonterminal Y, let $L(Y)$ be the set of all trees derivable from Y in G. Let $L(G) := \bigcup_{Y \in N(G)} L(Y)$. For a production P, let $n(P)$ be the nonterminal that appears on the left-hand side of P. Let $G(Y) := \{P \in G : n(P) = Y\}$ be the set of productions from Y in G. Throughout, we will use M to denote $|G|$, N to denote $|N(G)|$, R to denote the maximum size of a production and L to denote the total length of all productions (in symbols).

A *derivation tree* T a tree produced by the grammar G that is labelled with the productions used in its production. We will denote by $r(T)$ the production which labels the root. We will denote by $P(T_1, \ldots, T_t)$ the tree whose root is labelled with P and where the children of the root are the subtrees $T_1, \ldots, T_t$.

Cost Orders

A *pre-order* (sometimes called a *quasi-order*) on S is a binary relation $\preceq$ on $S \times S$ with the following properties:

1. (reflexivity) $\forall s \in S$, $s \preceq s$
2. (transitivity) $\forall u, v, w \in S$, if $u \preceq v$ and $v \preceq w$, then $u \preceq w$

A pre-order is *total* (sometimes called *linear*) if for any s and t in S, it holds that $s \preceq t$ or $t \preceq s$. Note that there could be distinct s and t in S such that $s \preceq t$ and $t \preceq s$ (we write this as $s \approx t$).

We define a *cost-order* on terms as any total pre-order on the set $T(\Sigma) \cup \{\infty\}$ for a ranked alphabet Σ. The element ∞ has the property that for all $t \in T(\Sigma)$, it holds that $t \preceq \infty$.

A cost-order is called *monotone* if for all $t_1, \ldots, t_r$ and t_i' in $T(\Sigma)$ and all functions $g \in \Sigma$, it holds that if $t_i \succeq t_i'$ then $g(t_1, \ldots, t_i, \ldots, t_r) \succeq g(t_1, \ldots, t_i', \ldots, t_r)$. Intuitively this means that increasing the cost of a subterm can only make the whole term cost more.

A cost-order is called *non-decreasing* if for all $t_1, \ldots, t_r$ in $T(\Sigma)$ and all functions $g \in \Sigma$ it holds that $g(t_1, \ldots, t_r) \succeq t_i$ for all i. Intuitively this means that a term can never cost less than its subterms.

A cost-order that is both monotone and non-decreasing is called *superior*. The enumeration of terms according to superior cost-orders was studied in [15].

In our work, we replace the non-decreasing requirement with the more general *weakly non-decreasing* requirement. A cost-order is *weakly non-decreasing* if $t_1, \ldots, t_r$ in $T(\Sigma)$ and all functions $g \in \Sigma$, whenever $g(t_1, \ldots, t_i, \ldots, t_r) \preceq t_i$ it then follows that $g(t_1, \ldots, t_i, \ldots, t_r) \approx g(t_1, \ldots, \infty, \ldots, t_r)$. In other words, terms can have smaller cost than their subterms as long as replacing that subterm with ∞ does not change the cost.

Cost Functions

The original work by Knuth did not discuss orders on grammars. Instead, it focused on functions from terms to the real numbers. While less general than the cost-orders defined above, it can be intuitive to consider costs as the output of functions on terms.

Knuth specifically considered the set $\mathcal{R}^*$ defined as the non-negative real numbers with a special ∞ symbol, but any cost $\mathcal{D}$ with corresponding total pre-order will work. Each function symbol f can be assigned a cost-function c_f defined on elements in $\mathcal{D}$. For each term $t = f(t_1, \ldots, t_r)$ the cost $c(t)$ is defined recursively by $c(f(t_1, \ldots, t_r)) = c_f(c(t_1), \ldots, c(t_r))$.

For example, to represent term-depth using cost-functions, each constant a is given cost $c_a = 1$, and each function f is given cost $c_f(X_1, \ldots, X_r) = max\{X_1, \ldots, X_r\} + 1$. The cost of $f(a, g(a))$ would then be $c_f(c_a, c_g(c_a))) = c_f(1, max\{1\} + 1) = max\{1, 2\} + 1 = 3$.

We define a cost-function c_f to be *weakly superior* if it is monotone nondecreasing and for all $x_i \in \mathcal{D}$, $j \in [r]$, whenever $c_f(x_1, \ldots, x_r) < x_j$ it follows that $c_f(x_1, \ldots, x_r)$ equals $c_f(x_1, \ldots, x_{j-1}, \infty, x_{j+1}, \ldots, x_r)$.

From these we get the following easy remark.

Remark 2. *For any set $\mathcal{D}$ with total pre-order $\leq$ and cost functions for each symbol in Σ, let $\preceq$ be the order defined by $t \preceq s$ iff $c(t) \leq c(s)$ for $s, t \in T(\Sigma)$. Then $\preceq$ is weakly superior iff each cost function c_f for each $f \in \Sigma$ is weakly superior.*

Knuth's Algorithm

Knuth [39] gives an algorithm for finding a shortest derivation function for a superior RTG; Ramalingam and Reps [46] extend the result to weakly superior RTGs. The approach is a generalisation of the shortest path algorithm of Dijkstra.

Theorem 16 ([39, 46])**.** *There is an algorithm* `Knuth` *that computes a shortest derivation function for a weakly superior RTG G in time $O(M \log N + L)$.*

Note that the shortest derivation function is in general not unique. The function τ output by Knuth's algorithm has some special properties which do not necessarily follow from the fact that τ is a shortest derivation function. In particular, if we consider τ as a directed graph whose vertices are $N(G)$ and where (X, Y) is an edge if $\tau(X)$ has Y on its right hand side, this graph is a DAG. It hence induces a topological ordering on $N(G)$, which is a property we will rely upon later.

We will assume that Knuth's algorithm, on input G, deterministically computes some mapping τ; in subsequent sections, we will use τ to refer to this mapping in particular. We will write $T^*_Y := T_{\tau,Y}$. Note that because the graph of τ is a DAG, T^*_Y is finite.

6.2 Enumeration Algorithm

We present a dynamic programming algorithm for enumerating the k shortest derivations in a weakly superior RTG.

Theorem 17. *Given a weakly superior RTG G, Algorithm 10 runs in time $O((M + kNR) \log(M + kNR))$ and outputs a data structure from which the k shortest derivations from any nonterminal can be obtained in time linear in their size.*

We say that a set S of derivations from Y is *complete* if for all derivations T, T' from Y with $v(T) < v(T')$, if $T' \in S$ then $T \in S$. Note that a complete set of size j contains 'the j shortest productions' with respect to v, for some choice of tiebreaking rule. Let $F(Y, [j]) := \{F(Y, i) : i \leq j\}$. The correctness of the algorithm is implied by the following lemma.

Lemma 11. *At the end of iteration (j, Y), $F(Y, [j])$ is complete, and $F(Y, j) = \bot$ if and only if $|L(Y)| < j$.*

Proof. We prove this by induction. For the base case, note that $F(Y, 1) = T^{(1)}_Y$ for all Y. Then suppose that for all X, i such that either $i < j$ or $i = j$ and $X < Y$, $F(X, [i])$ is complete, and $F(X, [i]) = \bot$ if and only if $|L(X)| < i$, at the end of iteration (i, X). By the structure of the algorithm, this happens before iteration (j, Y).

In particular, $F(Y, [j-1])$ is complete; if $\bot \in F(Y, [j-1])$ then $|L(Y)| < j - 1$ and we're done, so suppose otherwise. Then $F(Y, [j])$ does not satisfy the lemma only if for some derivation $T \notin F(Y, \{1, \ldots, j-1\})$ from Y, T' is added in iteration (j, Y) with $v(T') > v(T)$ or no derivation is added at all. This implies, in both cases, that $T \notin H_Y$ at the end of iteration (j, Y).

Let $T = P(T_1, \ldots, T_s)$. We will find some T'' with $v(T'') \leq v(T)$ but $T'' \in H_Y \setminus F(Y, [j-1])$, which is a contradiction because either the algorithm chose T' or emptied H_Y. First, for any

Input : A RTG G and bound $k \in \mathbb{N}$.
Output : A data structure representing the k shortest derivations from every nonterminal in G.

$\tau \leftarrow \texttt{Knuth}(G)$
compute topological ordering $<$ of $N(G)$ according to τ
for $Y \in N(G)$ *ordered by* $<$ **do**
  set $F(Y, 1) \leftarrow T_Y^{(1)}$
  initialize empty min-heap H_Y
  for $P \in G(Y)$ **do**
    $H_Y.\texttt{push}(P(T_{Z_1}^{(1)}, \ldots, T_{Z_t}^{(1)}))$
for $j = 2, \ldots, k$ **do**
  for $Y \in N(G)$ *ordered by* $<$ **do**
    $T = P(T_1, \ldots, T_t) \leftarrow F(Y, j-1)$
    for $\ell = 1, \ldots, t$ **do**
      let i_ℓ be such that $T_\ell = F(Z_\ell, i_\ell)$
      if $F(Z_\ell, i_\ell + 1) \neq \bot$ **then**
        $T'_\ell \leftarrow F(Z_\ell, i_\ell + 1)$
        $H_Y.\texttt{push}(P(T_1, \ldots, T_{\ell-1}, T'_\ell, T_{\ell+1}, \ldots))$
    repeat
      $T' \leftarrow H_Y.\texttt{pop}()$
    until $T' \notin F(Y, *)$ *or* H_Y *is empty*
    if $T' \notin F(Y, *)$ **then**
      $F(Y, j) \leftarrow T'$
return F

Algorithm 10: Algorithm for enumerating the k shortest derivations in a weakly superior RTG.

$T_\ell \notin F(Z_\ell, [j_0])$, we replace T_ℓ with $F(Z_\ell, j_0) \neq \bot$; note that this cannot increase $v(T)$ because $F(Z_\ell, [j_0])$ is complete, nor can it place T in $F(Y, [j-1])$. If this places T in H_Y, we are done.

Note now that for all ℓ, $T_\ell = F(Z_\ell, i_\ell)$ for some $i_\ell \leq j_0$. We now iterate the following procedure: while $T \notin H_Y$, find the least ℓ such that $P(T_1, \ldots, T_{\ell-1}, F(Z_\ell, i_\ell - 1), T_{\ell+1}, \ldots, T_s) \notin F(Y, [j-1])$, and set $i_\ell \leftarrow i_\ell - 1$. When $T \notin H_Y$, such an ℓ is guaranteed to exist by the structure of the algorithm. This procedure does not place T into $F(Y, [j-1])$, and by monotonicity, it cannot increase $v(T)$. Since $P(T_{Z_s}^{(1)}, \ldots, T_{Z_s}^{(1)}) \in F(Y, [j-1])$, the procedure must terminate with $T \in H_Y$. This proves the lemma. □

Hence at the end of the algorithm, for all nonterminals Y, $F(Y, k)$ is complete and contains $\max(k, |L(Y)|)$ distinct derivations; this establishes correctness. For the time bound, we assume that heap operations take logarithmic time and that trees can be compared for equality in constant time. The number of heap operations is at most

$$2\Big(|G| + k \sum_{Y \in N(G)} \max_{P \in G(Y)} |P|\Big) = O(M + kNR) \ ,$$

from which the time bound follows.

6.3 Discussion

In this section we present a few notes about the optimality and usefulness of our algorithm.

Weakly Superior Cost Functions

The algorithm presented here requires that the cost-orders be weakly superior. We first consider two relaxations of this requirement. These discussions already apply to the easier problem of finding a (single) shortest derivation.

Non-monotone superior cost functions If we place no restriction on cost functions, the problem of finding a shortest derivation becomes NP-complete. Indeed, fix the grammar $X \to f(X_1, \ldots, X_n), X_i \to 0|1$. For a given SAT formula ϕ, we set the cost function $f(b_1, \ldots, b_n)$ to be $\phi(b_1, \ldots, b_n) + 1$ (with $f(b_1, \ldots, b_n) = \infty$ for $b \notin \{0, 1\}$), and assign bit b cost b. Then the shortest derivation has value 1 if ϕ is satisfiable, and 0 otherwise. Note that this is a 'non-monotone superior' cost function, in that $f(b_1, \ldots, b_n) \geq \max(b_1, \ldots, b_n)$.

Monotone non-superior cost functions If the only restriction on our cost functions is that they are monotone, note that a special case of the problem of finding a shortest derivation from a RTG is finding the shortest path in a graph with negative edge weights. The best known algorithm for this problem is Bellman-Ford, which runs in time $\Theta(|V| \cdot |E|)$; considering the graph as a RTG, this is $\Theta(MN)$. Hence if we were able to remove the superiority condition without affecting the running time of our algorithm, we would be able to find shortest paths faster than Bellman-Ford (k and R are both constant in this case). There is some evidence that this is impossible. [36]

We now give some evidence that weakly superior cost functions are sufficiently expressive for many applications, by showing that some natural cost functions and RTGs are weakly superior.

Program size The total number of function symbols (including constants) in a term.

Program depth The depth of a term when viewed as a tree (e.g., depth of a is 0, depth of $g(g(a), c)$ is 2).

Positive linear functions Any cost function of the form $\sum_i c_i X_i + d$ for $c_i, d \in \mathbb{R}^*$ is superior.

Non-decreasing polynomial functions Let p be an n-variate polynomial with all coefficients greater than 1. Then $p(\max(X_1, 1), \ldots, \max(X_n, 1))$ is superior.

Degree of polynomials A simple grammar representing n-variate polynomials is $V \to x_1 \mid \cdots \mid x_n \mid c$, $F \to V + V \mid V \cdot V$. The cost function can then represent the total degree of the polynomial: productions $V \to x_i$ have cost 1, $V \to c$ (for c in the ring of coefficients) has cost 0, $+$ has cost $\max(v_1, v_2)$ and $\cdot$ has cost $v_1 + v_2$. This is easily seen to be a superior RTG.

Round up to nearest multiple of constant Univariate functions of the form $c(\lfloor X/c \rfloor + 1)$, which rounds X up to the nearest multiple of some constant c. Can be used, for example, in counting memory usage when memory can only be allocated in increments of 8-bit bytes. So storing 10 bits would require the allocation of 16 bits (2 bytes) of memory.

Round up to nearest power of constant Univariate functions of the form $c^{\log_c(X)+1}$, which rounds X up to the nearest power of some constant c. Can be used, for example, in tracking memory usage in C++ vectors, which double memory allocation whenever an array is filled [55].

Finite resources Univariate functions which return X if $X \leq t$ and ∞ otherwise. Can be used to represent cases when there is a fixed number t of a given resource that can be used.

Constant Functions Any cost function that returns the same value regardless of input is weakly superior.

Minimum The function $min(X_1, \ldots, X_n)$ is weakly superior.

Median The function $med(X_1, \ldots, X_n)$, which outputs the median of $\{X_1, \ldots, X_n\}$ is weakly superior.

Lexicographic Ordering

Weakly superior functions are closed under composition [46]. This means that for any weakly superior n-ary functions $g_1, ..., g_k$ and weakly superior k-ary function f, the following function h is weakly superior:

$$h(X_1, \ldots, X_n) = f(g_1(X_1, \ldots, X_n), \ldots, g_k(X_1, \ldots, X_n))$$

Under normal assumptions, the lexicographic product does not actually conserve superiority and the pointwise product does not conserve totality (the pointwise product of two total functions might not be total).

Counter-example to lexicographic ordering.

Let $f_1(x) = \lceil x \rceil$ and $f_2(y) = y$. Let $f_{1,2}(x, y) = (f_1(x), f_2(y))$ (the normal lexicographic function application). By the lexicographic ordering, $(1/2, 1) > (1/3, 2)$. However, applying the functions yields $f_{1,2}(1/2, 1) = (1, 1)$ which is less than $f_{1,2}(1/3, 2) = (1, 2)$. So giving a bigger input yields a smaller output, violating the monotone property.

The problem is that $f_1(1/2)$ equals $f_1(1/3)$, so while the input points have $(1/2, 1) > (1/3, 2)$, when the map is applied, the first value is equal, so the second is considered.

If we instead assume that the functions are monotonically *increasing*, we get conservation under lexicographic order.

The proof pointwise product does not conserve totality is easy. We could have functions to the natural numbers in both orders, but then $(1, 2)$ is incomparable to $(2, 1)$.

6.4 Handling Let-Expressions

Syntax-Guided Synthesis allows for the use of 'let-expressions' of the form $(let[z_1 = e_1, z_2 = e_2, ..., z_r = e_r]e_{r+1})$, where each z_i is a variable and each e_i is an expression (term) made from nonterminals, variables, and alphabet symbols. Consider, for example, the following grammar: $S \to (let[z_1 = U, z_2 = z_1 + z_1]z_2 + 1\ U \to 0|1|U + U$

Here, S yields a set of expressions that representing any odd number. Specifically, it yields one plus the sum of identical expressions from U. The let-expression is necessary to guarantee the expressions in the sum are identical.

Some languages specified using let-expressions, such as the one above, are not regular. We can convert any such grammar G into a regular tree grammar G_{nolet} with no let-expressions such that there is a bijection between the languages produced be each grammar. This bijection matches each term with another of the same cost. Thus enumerating the minimal-cost terms in the new grammar is sufficient to enumerate the minimal-cost terms in the original grammar.

To construct G_{nolet} from G, first add all nonterminals, terminals, function symbols, and let-free productions from G. Let e' be any let-expression and let $A \to e'$ be a production in G. Let $[A_1, A_2, ..., A_l]$ be the list of nonterminals appearing in e' (with duplicates). A new production $A \to f_{e'}(A_1, \ldots, A_l)$ is added to G_{nolet} along with the new function symbol $f_{e'}$. For expressions $a_1, \ldots, a_l$ produced from non-terminals $A_1, \ldots, A_l$, respectively, the cost of $f_e(a_1, \ldots, a_k)$ equals the cost of $e'[A_1 \leftarrow a_1, \ldots, A_l \leftarrow a_l]$.

The above example would yield the grammar with productions $S \to f_{e'}(U)$ and $U \to 0 \mid 1 | U + U$. Using expression size as the original cost, the cost function for $f_{e'}$ would be $c_{f_{e'}}(x) = c_+(c_+(x, x), c_1) = 2 \cdot x + 3$.

6.5 SyGuS with Big-O Complexity

Arguably the most common costs that are considered when writing programs are their runtime and space complexity. These costs are usually written in big-O notation as a function of the input sizes. In this chapter, we will show how to account for common complexities, such as runtime and space, when written in big-O notation.

Rather than show how to enumerate terms by increasing complexity, this chapter shows how to modify a given SyGuS grammar so that it only produces terms less than a given complexity. For example, this process could yield a grammar that only creates programs with runtime complexity $n \cdot log(n)$ or less. This new grammar could then be passed to a SyGuS solver. If the solver finds a satisfying program, that program would be guaranteed to have runtime less than $n \cdot log(n)$. This process can be applied to any cost-order with a "low number of costs", such as finding all programs with depth less than 5.

Cost-Orders with Big-O

One simple cost-ordering we might consider would be based on the big-O notation of the cost, and would treat all polylog costs as being the same. These would be considered along with constants and polynomials. So, the set of costs over a variable n might be:

$$n^0 < polylog(n) < n^1 < polylog(n) \cdot n^1 < n^2 < \ldots$$

Here n would be the size of the input to a function. $Polylog$ refers to any function that is made from logarithm applications and polynomials of those results (e.g., $log(n)$, $log^2(n)$, $log^2(log(n))$).

One of the reasons we consider $polylog$ as a single cost, rather than treating each polylogarithmic function separately, is the $log(n)$ function. This function might appear, for example, as the cost of popping an element off a heap. It is difficult because creates infinitely decreasing chains (i.e.,

$log(n) > log(log(n)) > log(log(log(n))) > \dots$). This means there might not be a smallest cost term from a grammar . The use of $polylog$ circumvents this by avoiding the infinite chain, since $polylog(polylog(n)) = polylog(n)$.

If a function has two inputs of size n and m, we'd get a more complicated ordering.

$$m^0n^0 < m^0polylog(n) < m^0n^1 < m^0n^2 < \dots$$

$$polylog(m)n^0 < polylog(m)polylog(n) < polylog(m)n^1 < polylog(m)n^2 < \dots$$

$$m^1n^0 < m^1polylog(n) < m^1n^1 < m^1n^2 < \dots$$

Here, m^0n^1 might be incomparable to m^1n^0 but both would be less than m^1n^1.

If we have multiple inputs, but have some guess as to the relationship in size between these inputs, we can collapse the costs down into a single total linear order again. For example, if we believe that $m \approx n^2$, then the cost m^2n^1 would be about $(n^2)^2n^1 = n^5$.

This technique would be applied to the input-markers to the SyGuS function. For example, consider this grammar:

$$S \to ITE(C, S, S) \mid x \mid y \mid 0 \mid 1 \mid S + S$$

$$C \to S \leq S \mid \neg S \mid S \wedge S$$

The input-markers here are x and y, meaning the grammar might produce the function $f(x, y) = ITE(x \leq y, x, y)$. Say the cost of this $f(x, y)$ were m^2n^1, where n and m were the sizes of x and y respectively. If we thought $m \approx n^2$ then the cost of $f(x, y)$ (i.e., m^2n^1) would be about $(n^2)^2n^1 = n^5$.

In order to properly track the runtime of a function, we need to track the output sizes of subfunctions. Take for example the following grammar with start symbol $List$:

$$List \to sort(List) \mid makeList(Elt) \mid x$$

$$Elt \to makeElt(List) \mid pop(List)$$

The function $f(x) := sort(makeList(pop(x)))$ could come from the above grammar. This function takes the first element of a list, treats it as a list itself, and sorts it. So $f([1, 2, 3])$ would yield the list $[1]$. The runtime for $sort()$ would be $polylog(X) \cdot X$, where X is size of the input to $sort()$. However, in the function f, $sort$ is always given a list of constant size. It does not make sense to calculate the runtime of this function as a function of the size of the input x.

The solution is to track the size of the output for each subterm, as well as the runtime up to that point. So for each term s, the cost of s would be a pair $(size(s), time(s))$. The runtime for $sort(s)$ would be the runtime for calculating s plus the runtime for running $sort$ on the output of s, or $time(s) + polylog(size(s)) \cdot size(s) = \max\{time(s), polylog(size(s)) \cdot size(s)\}$ (for big-O notation, addition is equivalent to max). The output size of $sort(s)$ would be the same as the output size for s, which is $size(s)$ (sorting lists does not change their size). The output for $pop(s)$ would be $(const, const)$.

Applying this reasoning to the term $f(x) = sort(makeList(pop(x)))$ would yield the costs $size(f(x)) = const$ and $time(f(x)) = const$.

Modifying the SyGuS Grammars

Most of the big-O costs that we care about in practice are fairly small (e.g., n^2 or $nlog(n)$) and there are very few costs less than them according to the order described (there are only 4 costs less than n^2). We can take advantage of this to expand a given SyGuS grammar to account for big-O cost.

More specifically, given a tree grammar where all the functions have big-O cost as described above, we can create a new tree grammar (called the *cost-grammar*) such that the only programs that can be generated by the grammar have runtime or space usage less than a given cost (called the *target cost*). I'll use runtime as an example here, but everything also applies to space (or to minimizing over both runtime and space).

For example, consider this grammar over lists:

$$List \rightarrow sort(List) \mid append(List, Int) \mid [\,] \mid x$$
$$Int \rightarrow pop(List) \mid Int + Int \mid 0 \mid 1$$

Each function has the most natural cost associated with it as a function of its input. For example:

$$size(append(X, Y)) := size(X)$$

$$time(append(X, Y)) := const$$

Say we want to search for possible programs that are at most linear in $size(x)$ (define $n := size(x)$). We can create a new grammar where nonterminals are labelled based on the size of their output. So the nonterminal $List[n]$ would produce all programs yielded from $List$ above which have output size less than or equal to $O(n)$. Performing the actual conversion is not very difficult. Just assemble all possible nonterminal-cost pairs less than the target cost and check which functions map to what size outputs.

Applying this conversion to the above grammar yields:

$$List[const] \rightarrow sort(List[const]) \mid append(List[const], Int[const]) \mid [\,]$$
$$List[n] \rightarrow append(List[n]) \mid List[const] \mid x$$
$$Int[const] \rightarrow pop(List[n]) \mid Int[const] + Int[const] \mid 0 \mid 1$$

In this grammar, note that $sort$ is allowed on the outputs of $List[const]$, since sorting a constant-size list takes constant time. It is not allowed on outputs of $List[n]$ since that would take time $n \cdot polylog(n)$, which is greater than target-cost.

The nonterminals, $List[polylog(n)]$, $Int[polylog(n)]$ and $Int[n]$ are not included, since they yield equivalent programs to one of the above nonterminals. E.g., $Int[n]$ would yield the same set of programs as $Int[const]$, since all integer-yielding programs yield outputs of size $const$.

Notice that we only need to track the output size from each nonterminal, not the runtime (or space usage). This is because runtime and space are both non-decreasing. If a subprogram (i.e., subterm) takes time, say, n^2, then the larger program must also take time at least n^2. So it is enough to construct the cost-grammar so that every program that is generated from any nonterminal has runtime (or space) less than the target cost.

Chapter 7

Conclusion and Future Work

This thesis has presented several results related to program synthesis and exact active learning. We have described the first algorithms for learning equational theories. We've shown how modular programming can be used to learn the cross-products of concept spaces. Decidability results for SyGuS have been described. And finally, we've demonstrated a new way to run SyGuS with respect to costs. Each of these results raises open questions and new directions.

Part one, which focuses on concept learning, raises many new questions on which concept classes to learn and new techniques for learning. Having presented the first work on learning equational theories, there remain many classes of equational theories for which active learning algorithms can be developed. Non-collapsing shallow theories only allow variables at depth one, but different restrictions on the placement of variables in equations may lead to more interesting classes of learnable equational theories. There is also a very close connection between equational theories and EUF, and learning algorithms for equational theories might be adapted to learning EUF specifications. The work on modular learning answered questions on learning cross-products for some small sets of queries, but there are many more combinations of query-sets for which learning reductions are undecidable. There are also many more ways besides cross-products to combine concepts, which might be studied.

Part two raises new questions on SyGus decidability and ways to account for program costs. It remains open whether SyGuS is decidable modulo linear integer arithmetic and linear real arithmetic. We presented an enumeration algorithm for SyGuS programs according to weakly superior cost orders. However, the state-of-the-art for SyGuS solvers involves additional techniques in order to use enumeration as a synthesis tool. Future work will account for these techniques. We will also understand more about what programs can be synthesized with respect to big-O notation. The thesis used lists as an example, but it might be possible to synthesize programs from a more robust programming language that can describe general data structures and types. This new way of representing costs could yield significant improvements to the creation of more efficient programs.

Finally, as neural networks play a more important role in our world, there is much exciting work to be done on modeling them for explanations and verification. It is my belief that techniques from program synthesis, particularly OGIS, can be used to model neural network behavior as programs, making them safer and more understandable.

Bibliography

[1] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. "Scaling enumerative program synthesis via divide and conquer". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2017, pp. 319–336.

[2] Rajeev Alur et al. "SyGuS-Comp 2016: Results and Analysis". In: *Proceedings Fifth Workshop on Synthesis, SYNT@CAV 2016, Toronto, Canada, July 17-18, 2016*. 2016, pp. 178–202. DOI: 10.4204/EPTCS.229.13. URL: http://dx.doi.org/10.4204/EPTCS.229.13.

[3] Rajeev Alur et al. "Syntax-Guided Synthesis". In: *Dependable Software Systems Engineering*. 2015, pp. 1–25. DOI: 10.3233/978-1-61499-495-4-1. URL: https://doi.org/10.3233/978-1-61499-495-4-1.

[4] Rajeev Alur et al. "Syntax-guided synthesis". In: *Formal Methods in Computer-Aided Design (FMCAD), 2013*. IEEE. 2013, pp. 1–8.

[5] Dana Angluin. "Learning regular sets from queries and counterexamples". In: *Information and computation* 75.2 (1987), pp. 87–106.

[6] Dana Angluin. "Queries and concept learning". In: *Machine learning* 2.4 (1988), pp. 319–342.

[7] Hiroki Arimura, Hiroshi Sakamoto, and Setsuo Arikawa. "Learning Term Rewriting Systems from Entailment." In: *ILP Work-in-progress reports*. 2000.

[8] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge university press, 1999.

[9] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. www.SMT-LIB.org. 2016.

[10] Clark Barrett et al. "Satisfiability Modulo Theories". In: *Handbook of Satisfiability*. Ed. by Armin Biere, Hans van Maaren, and Toby Walsh. Vol. 4. IOS Press, 2009. Chap. 8.

[11] Shai Ben-David, Nader H Bshouty, and Eyal Kushilevitz. "A composition theorem for learning algorithms with applications to geometric concept classes". In: *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. 1997, pp. 324–333.

[12] James Bornholt et al. “Optimizing synthesis with metasketches”. In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 2016, pp. 775–788. DOI: 10.1145/2837614.2837666. URL: https://doi.org/10.1145/2837614.2837666.

[13] Randal E Bryant, Steven German, and Miroslav N Velev. “Exploiting positive equality in a logic of equality with uninterpreted functions”. In: *Proceedings of Computer Aided Verification (CAV)*. Springer. 1999, pp. 470–482.

[14] Nader H Bshouty. “A new composition theorem for learning algorithms”. In: *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. 1998, pp. 583–589.

[15] Matthias Büchse et al. “n-Best Parsing Revisited”. In: *ATANLP@ACL 2010*. Association for Computational Linguistics, 2010, pp. 46–54.

[16] Jean-Marc Champarnaud et al. “Bottom Up Quotients and Residuals for Tree Languages”. In: *arXiv preprint arXiv:1506.02863* (2015).

[17] Alexander Clark and Franck Thollard. “PAC-learnability of probabilistic deterministic finite state automata”. In: *Journal of Machine Learning Research* 5.May (2004), pp. 473–497.

[18] H. Comon et al. *Tree Automata Techniques and Applications*. Available on: http://www.grappa.univ-lille3.fr/tata. release October, 12th 2007. 2007.

[19] Hubert Comon, Marianne Haberstrau, and J-P Jouannaud. “Decidable problems in shallow equational theories”. In: *Logic in Computer Science, 1992. LICS’92., Proceedings of the Seventh Annual IEEE Symposium on*. IEEE. 1992, pp. 255–265.

[20] Hubert Comon, Marianne Haberstrau, and Jean-Pierre Jouannaud. “Syntacticness, cycle-syntacticness, and shallow theories”. In: *Information and Computation* 111.1 (1994), pp. 154–191.

[21] David Cyrluk, Oliver Möller, and Harald Rueß. “An efficient decision procedure for the theory of fixed-sized bit-vectors”. In: *Proceedings of Computer Aided Verification (CAV)*. Springer. 1997, pp. 60–71.

[22] G. B. Dantzig and B. C. Eaves. “Fourier-Motzkin elimination and its dual”. In: *Journal of Combinatorial Theory A* 14 (1973), pp. 288–297.

[23] Anatoli Degtyarev and Andrei Voronkov. “The undecidability of simultaneous rigid E-unification”. In: *Theoretical Computer Science* 166.1 (1996), pp. 291–300.

[24] Nachum Dershowitz. “Synthesis by completion”. In: *Urbana* 51 (1985), p. 61801.

[25] Nachum Dershowitz and Uday S Reddy. “Deductive and inductive synthesis of equational programs”. In: *Journal of Symbolic Computation* 15.5-6 (1993), pp. 467–494.

[26] Bruno Dutertre. “Solving Exists/Forall Problems With Yices”. In: *Proceedings of the International Workshop on Satisfiability Modulo Theories (SMT)*. 2015.

[27] Jeanne Ferrante and Charles Rackoff. “A Decision Procedure for the First Order Theory of Real Addition with Order”. In: *SIAM Journal of Computing* 4.1 (1975), pp. 69–76.

[28] E Mark Gold. "Language identification in the limit". In: *Information and control* 10.5 (1967), pp. 447–474.

[29] Cordell Green. "Application of theorem proving to problem solving". In: *Readings in Artificial Intelligence*. Elsevier, 1981, pp. 202–222.

[30] Sumit Gulwani et al. "Synthesis of loop-free programs". In: *SIGPLAN Notices* 46 (6 June 2011), pp. 62–73.

[31] John E Hopcroft and Jeffrey D Ullman. *Introduction to automata theory, languages, and computation*. Pearson Education India, 1979.

[32] Qinheping Hu and Loris D'Antoni. "Syntax-Guided Synthesis with Quantitative Syntactic Objectives". In: *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*. 2018, pp. 386–403. DOI: 10.1007/978-3-319-96145-3_21. URL: https://doi.org/10.1007/978-3-319-96145-3%5C_21.

[33] Susmit Jha and Sanjit A Seshia. "A theory of formal synthesis via inductive learning". In: *Acta Informatica* 54.7 (2017), pp. 693–726.

[34] Susmit Jha et al. "Oracle-Guided Component-Based Program Synthesis". In: *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering (ICSE)*. 2010, pp. 215–224.

[35] Susmit Kumar Jha. "Towards Automated System Synthesis Using SCIDUCTION". PhD thesis. EECS Department, University of California, Berkeley, 2011. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-118.html.

[36] Stasys Jukna and Georg Schnitger. "On the optimality of Bellman-Ford-Moore shortest path algorithm". In: *Theor. Comput. Sci.* 628 (2016), pp. 101–109. DOI: 10.1016/j.tcs.2016.03.014. URL: https://doi.org/10.1016/j.tcs.2016.03.014.

[37] Michael J Kearns and Umesh Virkumar Vazirani. *An introduction to computational learning theory*. 1994.

[38] Donald E Knuth and Peter B Bendix. "Simple word problems in universal algebras". In: *Automation of Reasoning*. Springer, 1983, pp. 342–376.

[39] Donald E. Knuth. "A Generalization of Dijkstra's Algorithm". In: *Inf. Process. Lett.* 6.1 (1977), pp. 1–5. DOI: 10.1016/0020-0190(77)90002-3. URL: https://doi.org/10.1016/0020-0190(77)90002-3.

[40] Dexter Kozen. "Complexity of finitely presented algebras". In: *Proceedings of the ninth annual ACM symposium on Theory of computing*. ACM. 1977, pp. 164–177.

[41] Dexter Kozen. "On the Myhill-Nerode theorem for trees". In: *Bull. Europ. Assoc. Theor. Comput. Sci* 47 (1992), pp. 170–173.

[42] Z. Manna and R. Waldinger. "A deductive approach to program synthesis". In: *ACM TOPLAS* 2.1 (1980), pp. 90–121.

[43] Robert Nieuwenhuis. “Basic paramodulation and decidable theories”. In: *Logic in Computer Science, 1996. LICS’96. Proceedings., Eleventh Annual IEEE Symposium on*. IEEE. 1996, pp. 473–482.

[44] Michael J O’donnell. “Equational logic as a programming language”. In: (1985).

[45] José Oncina and Pedro Garcia. “Inferring regular languages in polynomial update time”. In: (1992).

[46] G. Ramalingam and Thomas W. Reps. “An Incremental Algorithm for a Generalization of the Shortest-Path Problem”. In: *J. Algorithms* 21.2 (1996), pp. 267–305.

[47] MRK Rao. “Learnability of term rewrite systems from positive examples”. In: *Proceedings of the 12th Computing: The Australasian Theroy Symposium-Volume 51*. Australian Computer Society, Inc. 2006, pp. 133–137.

[48] Sanjit A Seshia. “Combining induction, deduction, and structure for verification and synthesis”. In: *Proceedings of the IEEE* 103.11 (2015), pp. 2036–2051.

[49] Sanjit A Seshia. “Sciduction: Combining induction, deduction, and structure for verification and synthesis”. In: *DAC Design Automation Conference 2012*. IEEE. 2012, pp. 356–365.

[50] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

[51] Armando Solar-Lezama. *Program synthesis by sketching*. Citeseer, 2008.

[52] Armando Solar-Lezama et al. “Combinatorial Sketching for Finite Programs”. In: *SIGOPS Oper. Syst. Rev.* 40.5 (Oct. 2006), pp. 404–415. ISSN: 0163-5980. DOI: 10.1145/1168917.1168907. URL: http://doi.acm.org/10.1145/1168917.1168907.

[53] Armando Solar-Lezama et al. “Combinatorial sketching for finite programs”. In: *ACM Sigplan Notices* 41.11 (2006), pp. 404–415.

[54] Armando Solar-Lezama et al. “Programming by sketching for bit-streaming programs”. In: *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. 2005, pp. 281–294.

[55] *std::vector*. URL: http://www.cplusplus.com/reference/vector/vector/.

[56] Abhishek Udupa et al. “TRANSIT: specifying protocols with concolic snippets”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’13, Seattle, WA, USA, June 16-19, 2013*. 2013, pp. 287–296. DOI: 10.1145/2491956.2462174. URL: https://doi.org/10.1145/2491956.2462174.

[57] Abhishek Udupa et al. “TRANSIT: Specifying Protocols with Concolic Snippets”. In: *Proceedings of the 34^{th} ACM SIGPLAN conference on Programming Language Design and Implementation*. 2013, pp. 287–296.

[58] Aad Van Der Vaart and Jon A Wellner. “A note on bounds for VC dimensions”. In: *Institute of Mathematical Statistics collections* 5 (2009), p. 103.

[59] Maarten H Van Emden and Keitaro Yukawa. “Logic programming with equations”. In: *The Journal of Logic Programming* 4.4 (1987), pp. 265–288.

[60] Margus Veanes. *On computational complexity of basic decision problems of finite tree automata*. Tech. rep. UPMAIL Technical Report 133, Uppsala University, Computing Science Department, 1997.

[61] Margus Veanes. "The undecidability of simultaneous rigid E-unification with two variables". In: *Kurt Gödel Colloquium on Computational Logic and Proof Theory*. Springer. 1997, pp. 305–318.

[62] Gail Weiss, Yoav Goldberg, and Eran Yahav. "Extracting automata from recurrent neural networks using queries and counterexamples". In: *arXiv preprint arXiv:1711.09576* (2017).