Definability of Tree Languages

David Faig

June 11, 2018

Contents:

1. Introduction

The goal of this report is to introduce tree languages and their associated decision problems. We want a procedure that allows us to determine when a regular tree language is or is not first order definable. In the case of regular string languages there is a full solution, but for regular tree languages it is not so clear. We will cover the case of string languages and see how it compares to tree languages.

2. Regular String Languages

In computer science, regular expressions and strings are found in both theoretical and applied problems. Regular expressions are a common way to search text files, file systems and databases. For example, an expression like $\{a, b\} \cdot c$ will find me every occurrence of ac or bc. Problems related to computability and computational complexity are often formulated using string languages. For example, can a Turing machine decide if a string encodes a Turing machine? For this paper, we will focus on deciding which regular string languages are definable in first and second order logic.

For the rest of this section, Σ will be a finite alphabet. A string language is a set of strings over Σ. We call a string language regular if it can be formed by a regular expression. For this paper, we will only need the basics of regular expression syntax. We can construct the set of all regular languages using unions, concatenations, and the Kleene star operation. For more details on regular expressions, see page 44 of [6].

- The empty language, $\emptyset$, is a regular language.
- All singleton languages including the empty string language, $\{\epsilon\}$, are regular.
- Unions and concatenations ($+$ and $\cdot$ respectively) of regular languages are regular.
- If A is a regular language, then the smallest superset of A that is closed under concatenation and contains ϵ is regular. Call this A*.

Examples:

- Σ^* is the language containing all strings over Σ.
- $(\{a, b\} \cdot c)^*$ is the language $\{\epsilon, bc, ac, bcbc, bcac, acac, ...\}$.
- $\{a, b\} \cdot c^*$ is the language $\{\epsilon, a, b, bc, ac, bcc, acc, bccc, ...\}$.

We will see that there is an alternative way to define the regular languages in the next section.

3. Deterministic Finite State Automata

Another way to define a language is to find a machine that decides which strings are in the language. Such a machine would take an input string, complete a computation, then return *accept* or *reject*. The language accepted by a machine is the set of string inputs that return *accept*. It turns out that a certain type of machine can decide exactly when a language is regular.

A deterministic finite state automaton (DFA) is a tuple $A = (Q, \Sigma, q_0, F, \delta)$ such that:

- Q is a finite set of states.
- Σ is a finite alphabet.
- q_0 is the initial state ($q_0 \in Q$).
- F is the set of accepting states ($F \subset Q$).
- δ is a transition function $\Sigma \times Q \to Q$.

You can find a very detailed introduction to DFA's starting at Page 36 of [6].

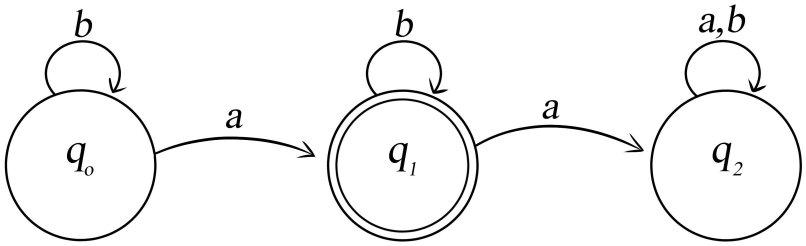

Figure 1: DFA

Note that A appears much like a directed graph where Q is the set of nodes, and δ is the set of edges. Given a list of instructions $[a_0, ..., a_n]$ from Σ, we may compute a "run" of $[a_0, ..., a_n]$ over A as follows: Start at q_0 and continue by computing $q_1 = \delta(a_0, q_0)$ followed by $q_2 = \delta(a_1, q_1)$ and so on. For shorthand, we can simply write $q_{i+1} = a_i(q_i) = a_i a_{i-1} ... a_0(q_0)$. We say that A **accepts** $[a_0, ..., a_n]$ if $a_n a_{n-1} ... a_0(q_0) \in F$.

Let $w = w_n w_{n-1} ... w_0$ be a Σ-string where each $w_i \in \Sigma$. We say A accepts w if A accepts $[w_0, ..., w_n]$ which can be conveniently be written as $w(q_0) \in F$. For each DFA, A, there is a language $L(A)$ of strings accepted by A. For example, the DFA in figure 1 accepts the language $b^* \cdot a \cdot b^*$.

Theorem 1 (Kleene): A language L is regular iff L is accepted by some finite state automaton.

Proof Sketch. For ($\Leftarrow$) induction on the size of Q works. The proof is much like induction on the size of graphs. By observing that subsets of Q can form automata that must accept only regular languages, we can deduce that A must only accept regular languages.

For ($\Rightarrow$) it helps to use nondeterministic finite state automata (NFA). An NFA is similar to a DFA except that the transition function $\delta : P(Q) \times \Sigma \to P(Q)$ transitions between sets of states. An NFA accepts a string w if $w(\{q_0\}) \cap F \neq \emptyset$ that is, if any state in the output is an accepting state.

The idea for the proof is to use induction on the recursive definition of regular languages. For example, if A is an NFA that accepts L then by mapping each state in F back to q_0 we can form a NFA that accepts L^*. $\square$

To see a full proof of Theorem 1, see page 109 of [4].

4. Defining String Languages

Finite Models:

A logical vocabulary σ is a collection of functions, relations, and constants. A σ-structure has a set called a universe, coupled with functions, relations, and constants that correspond to those in σ. A string w can be interpreted as a σ-structure $M_w = \langle A, <, F \rangle$ where A is a finite universe, $<$ is a linear order relation, and F is a set of unary predicates for each character in Σ. The predicates in F return whether a given element is a certain character. For a complete introduction to finite model theory, see [1].

Example 1.1: The string *apple* can be interpreted as $M_{apple} = \langle \{1, 2, 3, 4, 5\}, <, \{a, p, l, e\} \rangle$ where $a(1) = true$, $a(2) = false$, $p(2) = true$, $p(3) = true$, ... etc.

We can define languages using σ-structures in the following way: Let T be a set of logical sentences from the language of σ-structures. The language determined by T consists of all strings w such that $M_w \vDash T$.

Example 1.2: $\{\exists x\ a(x) \wedge \forall y(y \neq x \rightarrow x < y)\}$ defines the language of all strings starting with a.

First Order Logic:

A first order language (not to be confused with a string language) contains:

1. Countably many variable symbols.
2. Some (or zero) function symbols, relation symbols, and constants.
3. Propositional connectives such as $\wedge$, $\neg$, ... etc.
4. The quantifiers $\forall$ and $\exists$ which quantify over single variable symbols.
5. (optional) The equality symbol '='.

Given a string language L defined by the set of sentences T, we say L is first order definable if the logical sentences in T can all be written in a first order language. The first thing to note is that regular languages are not always first order definable.

Example 2: $(aa)^*$ is not first order definable.

Before we can prove the example we will need to introduce some definitions and tools.

The **rank** of a formula is the nesting depth of its quantifiers. For example,
$\forall x\ \exists y(y < x)$ has rank 2, while $\exists x(x = 1) \wedge \exists y(y = 2)$ has rank 1.

The **k-type** of a σ-structure A is the set of sentences with quantifier rank up to k that are satisfied by A.

A useful tool in model theory is the Ehrenfeucht-Fraisse (EF) game which allows us to decide whether a property of a structure can be expressed in first order logic. Given some $k \geq 0$, the goal in an EF game is to show that two structures are elementarily equivalent (i.e. they satisfy all the same logical sentences) for sentences with rank at most k. There is a full chapter on EF games in [1].

To play a single k-game we must have two players, the spoiler and the duplicator, and two

σ-structures A and B. Throughout the game the players construct a map $h : A' \to B'$ where A' and B' are substructures of A and B. The game proceeds as follows:

1. Both A' and B' start with empty universes.
2. The spoiler chooses A or B. Without loss of generality, assume they choose A.
3. The spoiler chooses an element a in A and adds it to A'.
4. The duplicator chooses an element b in B and adds it to B', declaring that $h(a) = b$.
5. Steps 3 and 4 repeat (without choosing the same element twice) until h is not an isomorphism or until $|B'| = k$.
6. If h is not an isomorphism, then the spoiler wins. Otherwise the duplicator wins.

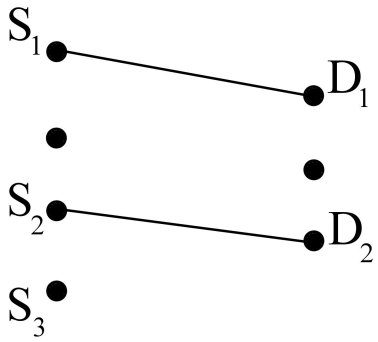

Figure 2: A 3-game where each side is an ordered set.

The game in figure 2 shows the duplicator's turn where they cannot make a choice without immediately losing. In this example, the spoiler can win every game no matter how the duplicator plays.

If the duplicator has a winning strategy for A and B (i.e. they can win every k-game), then we say A and B are **k-equivalent** written $A \equiv_k B$. Ehrenfeucht-Fraisse proved that two k-equivalent structures have the same k-types.

Proof of Example. We want to show that for every $k \in \mathbb{N}$ there are σ-structures A and B such that $A \equiv_k B$ and $A \in (aa)^*$ while $B \notin (aa)^*$.

For a given $k \geq 1$, let A be a string of length k and B be a string of length $k + 1$. Let both A and B only contain the character a. Then either A or B is in $(aa)^*$ while the other is not.

Since A and B only have the character a, the duplicator only needs to worry about order. Since the spoiler can only pick at most k characters, the duplicator will always have enough choices to preserve order. Hence, $A \equiv_k B$.

Suppose some logical sentence Φ defines $(aa)^*$. Then if k is the quantifier depth of Φ, we can find A and B as above. By assumption, $A \in (aa)^*$ which means $A \vDash \Phi$, but since $B \equiv_k A$ we have $B \vDash \Phi$ which a contradiction since $B \notin (aa)^*$. $\square$

5. Monadic Second Order Logic:

Monadic second order logic (MSO) is an extension of first order logic that allows quantification over sets (but not predicates). Here we will see that it is exactly the amount of power we need to define all regular string languages.

For clarity, all capitalized variables are **set-variables** (second order variables are arity 1). All set-variables are unary predicates that decide set inclusion. More precisely, given a set-variable X and a constant a, we have $X(a) = true$ iff $a \in X$. For this paper, we will use $X(a)$ and $a \in X$ interchangeably.

Theorem 2 (Büchi): A string language is MSO definable iff it is regular.

To find a complete proof of this theorem, see page 124 of [1].

Proof sketch ($\Leftarrow$). Let L be the language accepted by the DFA $A = (\{q_0, ..., q_m\}, \Sigma, q_0, F, \delta)$. In order to express a DFA operating on a list of instructions, we would like to quantify over each state. Here we can use quantification over sets to describe states in terms of string indices. The formula would look something like:

$$\exists X_0 ... \exists X_m \Phi(X_0 ... X_m)$$

Where each X_i represents the set of string indices that land on the q_i, and Φ encodes a DFA reading $X_0 ... X_m$ and ending on an accepting state. $\square$

Lemma 1: The language of σ-structures has finitely many k-types, and each k-type is finite (up to logical equivalence).

Proof of Lemma. It suffices to show that there are finitely many rank-k sentences (up to logical equivalence). This follows from the fact that there are finitely many functions and

relations in the language of σ-structures. Hence, there are finitely many atomic sentences, which implies that there are finitely many rank-0 sentences. By similar reasoning, there are finitely many rank-0 formulas that take exactly m free variables. Every rank-$k+1$ sentence is a boolean combination of sentences of the form $\exists x\phi(x)$ where $\phi(x)$ is a rank-k formula. By induction, there are finitely many rank-k sentences.

k-types are subsets of the set of rank-k sentences, so the lemma follows directly from the above proof. There is a more detailed proof of Lemma 1 on page 34, [1].

Proof sketch ($\Rightarrow$). Let Φ be a rank-k MSO sentence in the language of σ-structures. The idea for the proof is to construct a DFA by treating each k-type as a state (we need the lemma to justify doing this).

To define $\delta(q, w)$, find a string a such that M_a has k-type q, then return the k-type of $M_{a \cdot w}$. Lemma 2.1 in the next section will show that this function is well defined. $\square$

Example 3: $(aa)^*$ is MSO definable by the following sentence:

$$\forall x(a(x)) \wedge \exists X\, \exists Y\, \varphi_{part}(X, Y) \wedge \varphi_{odd}(X, Y) \wedge \varphi_{even}(X, Y)$$

Where $\varphi_{part}(X, Y)$ encodes that X and Y form a partition of the indices in the string:

$$\forall x((x \in X \vee x \in Y) \wedge (x \in X \rightarrow x \notin Y))$$

$\varphi_{odd}(X, Y)$ encodes that for each index in X there is another index in Y that is one greater.

$$\forall x(x \in X \rightarrow \exists y(y \in Y \wedge y > x \wedge \forall z(z > x \rightarrow z \geq y)))$$

$\varphi_{even}(X, Y)$ similarly encodes that each index in Y is one greater than some index in X:

$$\forall y(y \in Y \rightarrow \exists x(x \in X \wedge y > x \wedge \forall z(y > z \rightarrow x \geq z)))$$

Since $\varphi_{even}(X, Y)$ cannot have the first index, this forces the first index to be in X. Hence, X has all the odd indices and Y has all the even indices, and since $\varphi_{odd}(X, Y)$ forces Y to have the last index, the length of the string must be even.

6. Monoids and String Languages

As mentioned in the introduction, our original goal is to find a process that decides

whether a language is first order definable. For the string case that we have been discussing, there is a theorem that will provide a complete solution by converting the problem to one about monoids.

For this section, we must first describe star free languages, aperiodic monoids and the syntactic monoid. The following definitions can also be found in [2].

Star-free languages are a subset of regular languages where we replace the * operation with set negation. There are fewer star-free languages than regular languages, but with set negation we still have languages such as $\neg\emptyset$ which is the same as Σ^*.

Example 4: $\neg(ab \cdot (\neg\emptyset))$ is all strings that do not start with ab.

A **monoid** is a set M coupled with an associative binary operation. All monoids have an identity element. In other words, monoids are rings without addition. Note that Σ^* is a monoid under string concatenation with the identity element ϵ. In particular, Σ^* is the largest monoid that can be generated by Σ (the free monoid).

A monoid M is **aperiodic** when for every element $m \in M$, there is some number t such that $m^t m = m^t$.

Let M and N be monoids. $h : M \to N$ is a monoid homomorphism if the following are true:

- $a, b \in M$ we have $h(a)h(b) = h(ab)$.
- $h(1_M) = 1_N$

We say a language $L \subset \Sigma^*$ is recognized by a monoid M if there is a homomorphism $h : \Sigma^* \to M$ and a subset $F \subset M$ such that $h(L) \subset F$ and $h(\Sigma^* \backslash L) \subset M \setminus F$. In other words, h maps L and L^c to distinct subsets of M.

A monoid M is the **syntactic monoid** of L when it is the smallest monoid that recognizes L. It is smallest in the following sense: If L is recognized by the homomorphism $h : \Sigma^* \to M$, then any other monoid that recognizes L contains a submonoid that has a quotient that is isomorphic to $h(\Sigma^*)$. Note: By definition, the syntactic monoid is unique up to isomorphism.

Theorem 3: A language L is regular iff it is recognized by a finite monoid.

An equivalent theorem proved on page 115 of [4] states that L is accepted by some DFA iff the syntactic monoid is finite. By Theorem 1, the left side is equivalent to L being regular. The right side is equivalent since any finite monoid recognizing L must be "larger" than the syntactic monoid.

Theorem 4 (Schützenberger): Given a regular language $L \subset \Sigma^*$, the following are equivalent:

1. L is star-free.
2. L is first order definable.
3. The syntactic monoid of L is aperiodic.

To find all the background needed to understand the theorem as well as complete proofs of the theorem, see [2].

Proof. $(1 \Rightarrow 2)$: This can be done by induction on star-free expressions. For the base case, we show that the empty language and singleton languages are FO definable. For the induction step, we need to check that all the star-free operations preserve FO definability.

$\emptyset$ can be defined by false. A singleton set such as $\{a\}$ can be defined by:
$\exists x \forall y (x = y) \wedge a(x)$. Unions and complements of FO definable languages can be defined by taking disjunctions and negations respectively.

Concatenations require a little more work. Let L_1 and L_2 be first order definable by ϕ_1 and ϕ_2 respectively. To define $L_1 \cdot L_2$ we want to apply ϕ_1 to the L_1 part of each string and ϕ_2 to the L_2 part. This can be done by altering ϕ_1 into $\varphi_1(c)$ so that each sub-formula with a quantifier $\exists x \psi$ is replaced with $\exists x (x \leq c) \wedge \psi$. Similarly, alter ϕ_2 into $\varphi_2(c)$ so that $\exists x \psi$ is replaced with $\exists x (x > c) \wedge \psi$. In other words, $\varphi_1(c)$ applies ϕ_1 to the substring consisting of the first c characters, and $\varphi_2(c)$ applies ϕ_2 to the remaining substring. Hence $\exists c \varphi_1(c) \wedge \varphi_2(c)$ defines $L_1 \cdot L_2$. $\square$

You can find a more detailed version of $(1 \Rightarrow 2)$ in page 127 of [1].

For $(2 \Rightarrow 3)$ we will need the following lemmas from [2]:

Lemma 2.1: Given $w \equiv_k w'$ and $v \equiv_k v'$ we have $vw \equiv_k w'v'$.

Lemma 2.2: Given a string w, the k-type of w^{2^k} is equal to the k-type of w^{2^k+1}.

Proof of Lemma 2.1. We can use Ehrenfeucht-Fraisse games for this proof. We know the duplicator has a winning strategy for $w \equiv_k w'$ as well as $v \equiv_k v'$, the idea is to use those strategies to win a game from vw to $v'w'$.

For example, if the spoiler picks a character in v then the duplicator can use the winning strategy for $v \equiv_k v'$ and similarly if the spoiler picks a character in w. We only need to confirm that the relations are preserved when comparing characters from v to w or v' to w'. Since the $<$ relation is the only non-unary predicate, we can see that the constructed map is a partial isomorphism. Hence, the duplicator has a winning strategy and $vw \equiv_k v'w'$. $\square$

Proof of Lemma 2.2. For this proof, we want to leverage the fact that an Ehrenfeucht-Fraisse game only has k steps. As before, the only way for the spoiler to win is to force the duplicator to run out of choices that adhere to the $<$ relation. If the spoiler starts with w^{2^k}, then the duplicator can just use the first 2^k words in w^{2^k+1}. If the spoiler starts with w^{2^k+1}, then the duplicator can win by using the strategy in the following example:

1. The spoiler picks any character in w^{2^k+1}.
2. The duplicator chooses the same character from one of the two middle w's. This splits the original string into $w^{2^{k-1}-1}$ and $w^{2^{k-1}}$.
3. If the spoiler picks a character to the right of their first choice, then the duplicator repeats step 2 on whichever half was to the right.

Note that w^{2^n-1} always splits into two $w^{2^{n-1}-1}$, so we can split w^{2^k} exactly $k-1$ times before the algorithm tries to split a single w. Since there are k steps in a game, we know that the string will be split at most $k-1$ times leaving the duplicator with at least one w between each of their previous choices. Hence, the duplicator has a winning strategy to show that $w^{2^k+1} \equiv_k w^{2^k}$. $\square$

Proof of $2 \Rightarrow 3$. For this proof, it suffices to show that L is recognized by any finite aperiodic monoid. This is because any submonoids, including the syntactic monoid, will be aperiodic.

Let M be the set of different k-types over σ. Since σ is finite, so is M (can be shown by induction on k). Let $h : \Sigma^* \to M$ map strings to their k-types. By Lemma 2.1, h is a monoid morphism which induces a monoid structure on M. By Lemma 2.2, the monoid M is aperiodic. Since L is first order definable, membership of a string w in L depends only on its k-type, $h(w)$. Hence, M is a finite aperiodic monoid that recognizes L through h. $\square$

Proof sketch for $3 \Rightarrow 1$. The idea for this proof is to show that given a morphism $h : \Sigma^* \to M$ where M is an aperiodic monoid, we can find a star free expression for the pre-image of each $m \in M$. By doing so, we can construct a star free expression for any language recognized by M.

In order to create the expression for the pre-image, we must use Green's relations which are equivalence classes based on prefixes, suffixes, and infixes. The rest of the proof involves showing that for a fixed $m \in M$ there are two star-free languages that map to equivalence classes containing m. Those languages can be used to construct a star-free language describing the pre-image of m as desired. $\square$

When given a regular language L, we originally had no clear procedure for determining whether L could be defined in first order logic. As shown earlier, we needed an entire EF-game proof just to show that $(aa)^*$ is not first order definable. The usefulness of the Schützenberger theorem is that finding the syntactic monoid and determining whether it is aperiodic can be done in a procedural way.

7. Tree Languages

A ranked alphabet Σ is a finite set where each character has an associated arity. In this way, characters in Σ will behave as tree nodes, and the arity tells us the number of children. A tree can be written out by treating characters as functions. For example, $a(b(c, c), c, c)$ is a tree where a has arity 3, b has arity 2 and c has arity 0. In this tree, the *root* (the top) has type a, and all the *leaves* (arity 0 nodes) have type c.

Let $T(\Sigma)$ be the set of finite trees over sigma. A **tree language** is any subset of $T(\Sigma)$. Some examples include all trees of size 4, or all symmetric trees, or all trees where each node has only one child.

Note that a $T(\Sigma)$ can be viewed as an algebra where the carrier is $T(\Sigma)$ and the functions are each character in Σ. The example above illustrates how the function a takes three trees to output a single tree. This could also be done in the string case, though each function would have arity 1.

Note that any string can be interpreted as a tree where each node has arity 1 or 0. For this reason, anything we prove about tree languages also applies to string languages.

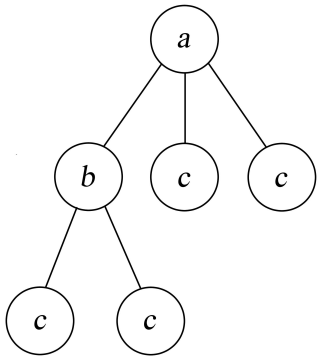

Figure 3: The tree a(b(c,c),c,c)

8. Recognizing Trees

While it was sufficient to recognize word languages with monoids, this may not be the case for trees. We will define recognition for trees more generally in the language of universal algebra. The following definitions can be found in [3].

Let $\mathbf{A} = (A, F)$ be an algebra with carrier A and functions F which correspond to the characters and their arities in Σ. We say $h : T(\Sigma) \to A$ is a homomorphism when the characters in Σ commute with the functions in F. I.e. for any $a \in \Sigma$, there is some $f_a : A^n \to A$ in F such that:

$$h(a(t_1, ..., t_n)) = f_a(h(t_1), ..., h(t_n))$$

We say that $\mathbf{A}$ recognizes L if there is a homomorphism $h : T(\Sigma) \to A$ where the image of L is disjoint from the image of $T(\Sigma) \setminus L$. Equivalently, $\mathbf{A}$ recognizes L if there is a set $H \subset A$ such that $h^{-1}(H) = L$. A tree language is **regular** if it is recognized by a finite algebra.

Given algebras $\mathbf{B} = (B, F_B)$ and $\mathbf{A} = (A, F_A)$:

1. $\mathbf{B}$ is called a reduct of $\mathbf{A}$ whenever $A = B$ and $F_B \subset F_A$.

2. $\mathbf{B}$ is called a subalgebra of $\mathbf{A}$ whenever $B \subset A$ and $F_B = F_A$.

3. $\mathbf{B}$ is called a quotient of $\mathbf{A}$ if there is a congruence relation $\sim$ on $\mathbf{A}$ such that $\mathbf{B} = \mathbf{A}/_\sim$.

4. We say $\mathbf{B}$ divides $\mathbf{A}$ whenever $\mathbf{B}$ is a quotient of a subalgebra of a reduct of $\mathbf{A}$.

A **term** over a language Σ is any function that can be formed by combining elements of Σ in the natural way.

A **polynomial** over Σ is a term that also allows constants. Note that constants are examples of polynomials. Let **polA** represent the algebra obtained from **A** by adding every polynomial to F. We say two algebras A and B are polynomially equivalent if **polA** is isomorphic to **polB**. To read more about polynomials, read page 9 of [3].

A **context** over Σ is a term over Σ with a single occurrence of a variable. For example, $a(b(c, x), c, c)$ is a context with x as the variable. A context p has an associated function $[p] : T(\Sigma) \to T(\Sigma)$ such that $[p](t)$ is the term p with t replacing the variable.

Theorem 5 (Myhill-Nerode): For every regular tree language L, there exists a finite algebra A that recognizes L such that A divides any other algebra that recognizes L.

There is a sketch for proving this theorem on page 7 of [3]. The sketch suggests that the following algebra is the syntactic algebra recognizing L:

For each context p associated sub-language of L defined by $\{t : [p](t) \in L\}$. This is called a derivative of L. Define the equivalence relation $\sim$ on $T(\Sigma)$ as: $t_1 \sim t_2$ iff t_1 and t_2 are in all the same derivatives of L. After showing that $\sim$ is a congruence on the algebra $T(\Sigma)$, we can form $T(\Sigma)/_\sim$ which is the syntactic algebra.

This theorem also proves the existence of the syntactic monoid for string languages. A context in the string case is simply a string where one of the characters is a variable that occurs once.

9. Defining Trees

For the rest of the paper, let the logical vocabulary σ have a unary predicate P_a for each $a \in \Sigma$, a binary predicate $<$ that returns whether the first node is a descendant of the other, and a binary predicate $child_i(a, b)$ that returns whether a is the "i'th child" of b for each $i > 0$. A tree t can be interpreted as a σ-structure M_t where the universe is the set of nodes in t. For this paper, every tree structure M_t will have ϵ_t as its root (or just ϵ when there is no need to distinguish).

Example 5: The tree in figure 3 can be interpreted as $M_t = \langle\{\epsilon, 1, 2, 3, 11, 12\}, \sigma^M, \rangle$ where σ^M is the interpretations of all the functions in σ.

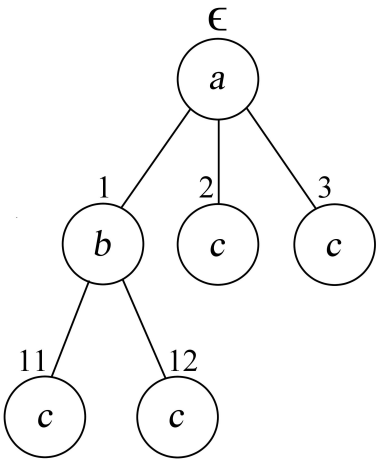

Figure 4: The tree from Example 5.

We can define tree languages in the same way we would define string languages by using sets of σ-sentences.

Example 6: The first order sentence $\exists x \forall y(x \neq y \rightarrow x < y)$ defines all trees where each node has arity 0 or 1 (all trees that form a single line).

As with regular string languages we want a way to determine whether a regular tree language is definable in first order logic. Unfortunately, regular tree languages are hard to categorize in first order. From theorem 3.2 in [3] we know that the syntactic algebra should determine whether a regular tree language is first order definable, but it is not clear how to extract this information. As shown in the example on page 10 of [3], the syntactic algebras of two languages can be polynomially equivalent while only one is definable in first order.

10. Tree Automata

Tree automata are related to DFA's except there are no starting states. Instead, computations start at the leaves of a tree's σ-structure. From there, the states of parent nodes are computed recursively until the root is reached where the final state is checked.

Formally, a (bottom-up) deterministic tree automaton $A = (Q, Q_f, \delta)$ over Σ is defined as follows:

As with DFA's, Q is the set of states and Q_f is the set of accepting states. The map $\delta : P(Q^*) \times \Sigma \to Q$ serves as "rules" for computing the states of parent nodes. That is, given a node v of type $a \in \Sigma$, if v has n child nodes that are mapped to the states $q_1, ..., q_n$ respectively, then v will be mapped to the state $\delta(q_1...q_n, a)$. Note that we only care about $\delta(s, a)$ when the length s is equal to the arity of a. For an arity 0 character $a \in \Sigma$, we will write $\delta(a)$ to avoid confusion with the empty string.

Given an input tree t, define a "run" of t on A as a function $r : M_t \to Q$ that adheres to the rules given by δ. We construct r by first mapping each leaf of M_t such that any leaf of type a gets mapped to $\delta(a)$. From there we map the parent nodes inductively according to δ until the root is reached. We say r is an accepting run if $r(\epsilon) \in Q_f$. We say A accepts t if there is an accepting run of t on A.

A nondeterministic tree automaton is similar to a deterministic tree automaton except $\delta : P(Q^*) \times \Sigma \to P(Q)$ outputs sets of states, meaning there can be multiple possible runs for a single tree. As long as one possible run is accepting, the input tree is accepted. For another definition of nondeterministic tree automata, see page 133 of [1].

Any language accepted by a nondeterministic tree automata is accepted by some determinisitc tree automata. To see a proof of this fact, see theorem 1.1.9 of [8], though that paper uses a far more technical definition of tree automata.

While tree automata function very differently from DFA's, they still correspond to every regular tree language. That is, theorem 1 is true for tree languages and tree automata:

Theorem 6: A tree language is regular iff it is accepted by a tree automaton.

Proof of ($\Rightarrow$). The idea for this proof is to convert an algebra into a nondeterministic tree automaton. Let $\mathbf{A} = (A, F)$ be an algebra that recognizes L with the homomorphism $h : T(\Sigma) \to A$. Construct the automaton (Q, Q_f, δ) as follows:

- Set $Q = A$.
- For any tree $t = b(t_1, ..., t_n)$, if the string $w = h(t_1)...h(t_n)$ then we have $h(t) \in \delta(w, b)$. In other words, δ directly corresponds to the homomorphism.
- Set $Q_f = h(L)$.

Given any tree $t \in L$ where $t = b(t_1, ..., t_n)$ for some $t_1, ..., t_n \in T(\Sigma)$, we can construct an accepting run r by working backward. To begin, $t \in L$ implies $h(t) \in Q_f$, so set $r(\epsilon_t) = h(t)$. By our construction, this gives us $h(t) \in \delta(h(t_1)...h(t_n), b)$, so we can set

$r(M_{t_i}) = h(t_i)$ for each $i = 1, ..., n$. We can repeat this process recursively until we reach the leaves. Hence, r is an accepting run. $\square$

Proof of ($\Leftarrow$). This proof has a similar idea, except it is much easier to work with a deterministic tree automaton when converting into an algebra. Let (Q, Q_f, δ) be a deterministic tree automaton that accepts a the tree language L. Construct the algebra (Q, F) such that for each character $a \in \Sigma$ of arity n there is a corresponding $f_a \in F$ of arity n such that $f_a(q_1, ..., q_n) = \delta(q_1...q_n, a)$. Let $h : T(\Sigma) \to Q$ be the homomorphism induced by mapping each arity 0 character $a \in \Sigma$ to $\delta(a)$.

Each tree t in L has a corresponding run r. For each leaf a in t, $h(a) = r(a)$ by how h was constructed. Since the functions in F are constructed to match δ, we have $r(\epsilon_t) = h(t)$. Hence, (Q, F) recognizes L since $h^{-1}(Q_f) = L$. $\square$

11. MSO and Chain Logic

Without much idea how to proceed in deciding whether regular tree languages are first order definable, we can try looking at other classes of regular tree languages. In particular, more is known about tree languages that are definable in MSO and in chain logic.

Theorem 7: A tree language is MSO definable iff it is regular.

The proof of this theorem is similar in spirit to theorem 2 except it uses tree automata instead of DFAs. That is, given a regular tree language, we have a corresponding tree automaton which we can then define in MSO. On the other hand, given an MSO expression that defines a tree language L, we can construct a tree automaton that accepts L by using k-types for states. For a full proof, see page 131 of [1].

Chain Logic

Chain logic is another extension of first order logic that allows quantification over chains. A chain is any set of nodes that are totally ordered by the descendant relation. In other words, a chain is any set of nodes lying on a single path to the root. Since chains are special types of sets, any language that is definable in chain logic is definable in MSO.

Given ranked alphabets Σ and Γ, a tree homomorphism is a map $h : T(\Sigma) \to T(\Gamma)$ such that for every character $a \in \Sigma$ there is a corresponding term $p_a \in \Gamma$ with the same

arity as a such that for any tree $t \in T(\Sigma)$:

$$h(a(t_1, ..., t_n)) = p_a(h(t_1), ..., h(t_n))$$

Note that if we interpret $T(\Gamma)$ as an algebra, then this definition is similar to finding a homomorphism from $T(\Sigma)$ to $\mathbf{pol}T(\Gamma)$.

The example on page 11 of [3] shows us that the class of first order definable tree languages is not closed under tree homomorphisms.

We found earlier that given a tree language L, membership in the class of first order definable tree languages is independent of $\mathbf{polA}$ where $\mathbf{A}$ is the syntactic algebra of L. This is not the case however for certain classes of tree languages described in theorem 4.3 of [3]. The theorem tells us that since the class C of languages definable in chain logic is closed under boolean combinations, inverse images of tree homomorphisms, and derivatives, then membership of L in C entirely dependent on $\mathbf{polA}$.

Conclusion

The problem of deciding whether a regular tree language is first order definable is still open. We have, however, found some insight into how the problem might be solved. We know that the syntactic algebra of a regular tree language should contain all the information we need. We were able to extract that information from the syntactic algebras of string languages, so there is hope that the same is possible for tree languages.

Bibliography

[1] Libkin, Leonid. *Elements of Finite Model Theory*, Springer, 2012.

[2] Bojańczyk, Mikołaj. (2014/2015) *Alegbraic Language Theory*, https://www.mimuw.edu.pl/ bojan/20142015-2/alg.

[3] Bojańczyk, Mikołaj; Michalewski, Henryk (2017). *Some Connections Between Universal Algebra and Logics for Trees*. arXiv:1703.04736

[4] Burris, Stanley; H. P. Sankappanavar. *A Course in Universal Algebra*, Springer-Verlag, 1981.

[5] Goldrei, Derek. *Propositional and Predicate Calculus: A Model of Argument*, Springer, 2005.

[6] Sipser, Michael. *Introduction to the Theory of Computation, 3rd Ed*, Sengage Learning, 2013.

[7] Regan, Kenneth (2007). *Notes on the Myhill-Nerode Theorem* https://cse.buffalo.edu/ regan/cse396/CSE396MNT.pdf

[8] Comon, Hubert; Dauchet, Max; Gilleron, Rémi; Jacquemard, Florent; Lugiez, Denis; Löding, Christof; Tison, Sophie; Tommasi, Marc (2008). *Tree Automata Techniques and Applications.*