Tree Automata

Mahesh Viswanathan

Fall 2018

Most machine models one studies in theoretical computer science, like Turing machines and finite automata, work on inputs that are strings over some finite alphabet. However, often problem descriptions have inputs that more structured than strings. For example, the input could be a graph, or a tree, or a partial order, etc. We will now introduce an automaton model that computes on inputs that are trees.

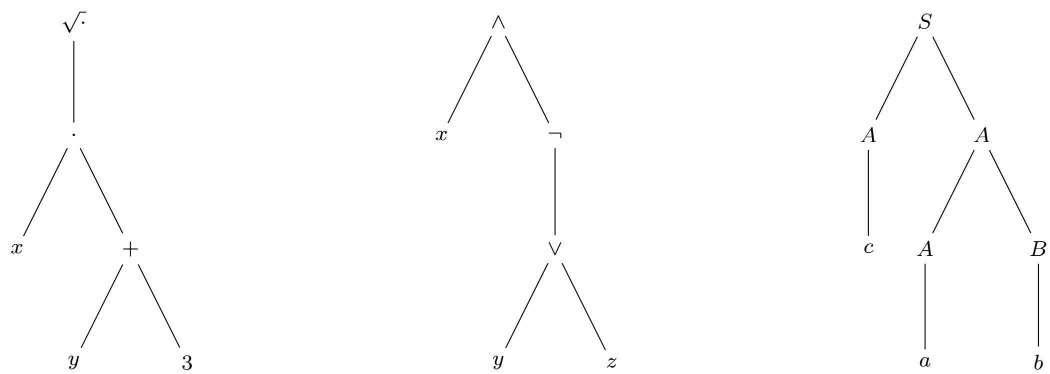

Figure 1: (a) Expression $\sqrt{x \cdot (y + 3)}$ (left); (b) Formula $x \wedge \neg(y \vee z)$ (middle); (c) Parse tree of grammar $S \to AA, A \to a \mid c \mid AB,\ B \to b$ (right).

Trees arise in a variety of contexts. Often strings semantically denote trees. Examples of this phenomena is shown in Figure 1. Arithmetic expressions or logical formulas maybe written as strings, but the scoping of the arithmetic and logical operators are best understood when one looks at the parse tree corresponding to the expression. Documents like XML and HTML are also textual representation of a tree, identified through the use of tags.

Trees are special kinds of graphs. Here we will consider *rooted trees*, i.e., trees where a special vertex has been designated as the root. There is a way to name vertices in a tree that makes explicit the parent/child and ancestor/descendent relation explict. Recall that in a rooted tree, every vertex has a unique path from the root. This unique path can be taken to be the name of vertices. For example, consider the tree shown in Figure 1a. The root is labeled $\sqrt{\cdot}$ is named ε; its child is 0 (labeled $\cdot$), and the other vertices are 00 and 01 (children of 0 and labeled x and $+$, respectively), 010 (labeled y), and 011 (labeled 3). We formalize this naming scheme and define labeled trees.

Definition 1. An n-ary *tree domain*, dom_t, is a prefix closed subset of $\{0, 1, 2, \ldots, n-1\}^*$ such that if $ui \in \mathrm{dom}_t$ then $uj \in \mathrm{dom}_t$ for every $j < i$.

A Γ-*labeled* n-*ary tree* is a pair $t = (\mathrm{dom}_t, \mathrm{val}_t)$, where dom_t is an n-ary tree domain and $\mathrm{val}_t : t \to \Gamma$ is a labeling function.

Example 2. Let define each of the trees in Figure 1 formally, as a labeled tree. Tree in Figure 1a is a

$\{\surd, \cdot, +, x, y, 3\}$-labeled tree $t_1 = (\{\varepsilon, 0, 00, 01, 010, 011\}, \mathrm{val}_{t_1})$, where

$$\begin{array}{lll} \mathrm{val}_{t_1}(\varepsilon) = \surd & \mathrm{val}_{t_1}(0) = \cdot & \mathrm{val}_{t_1}(00) = x \\ \mathrm{val}_{t_1}(01) = + & \mathrm{val}_{t_1}(010) = y & \mathrm{val}_{t_1}(011) = 3 \end{array}$$

Similarly, tree shown in Figure 1b is $t_2 = (\{\varepsilon, 0, 1, 10, 100, 101\}, \mathrm{val}_{t_2})$ with

$$\begin{array}{lll} \mathrm{val}_{t_2}(\varepsilon) = \wedge & \mathrm{val}_{t_2}(0) = x & \mathrm{val}_{t_2}(1) = \neg \\ \mathrm{val}_{t_2}(10) = \vee & \mathrm{val}_{t_2}(100) = y & \mathrm{val}_{t_2}(101) = z \end{array}$$

It is useful to introduce two operations on trees: the operation of building larger trees from smaller trees, and the operation of identifying subtrees of trees. Suppose t_0 and t_1 are Γ-labeled trees. Then, for any $A \in \Gamma$, $A(t_0, t_1)$ denotes tree with root labeled A, with left child of the root being the tree t_0, and the right child of the tree being t_1. This can be generalized to a tree of the form $A(t_0, t_1, \ldots t_k)$ formed from k trees $t_0 \ldots t_k$ with root labeled A. For a vertex/position p in a tree t, the subtree rooted at that vertex will be denoted by $t|_p$. We define these two operations precisely.

Definition 3. Given Γ-labeled trees $t_0 = (\mathrm{dom}_{t_0}, \mathrm{val}_{t_0})$ and $t_1 = (\mathrm{dom}_{t_1}, \mathrm{val}_{t_1})$, and $A \in \Gamma$, the tree $A(t_0, t_1)$ is given by $t = (\mathrm{dom}_t, \mathrm{val}_t)$ where

$$\mathrm{dom}_t = \{\varepsilon\} \cup \{0u \mid u \in \mathrm{dom}_{t_0}\} \cup \{1u \mid u \in \mathrm{dom}_{t_1}\}$$
$$\mathrm{val}_t(u) = \begin{cases} A & \text{if } u = \varepsilon \\ \mathrm{val}_{t_0}(v) & \text{if } u = 0v \\ \mathrm{val}_{t_1}(v) & \text{if } u = 1v \end{cases}$$

The above construction can be naturally generalized to the tree $A(t_0, \ldots t_k)$, $k \geq 0$, formed from $k+1$ trees $t_0, \ldots t_k$.

Given a Γ-labeled tree $t = (\mathrm{dom}_t, \mathrm{val}_t)$ and vertex/position $p \in \mathrm{dom}_t$, subtree rooted at position p, is the tree $t|_p = (\mathrm{dom}_{t|_p}, \mathrm{val}_{t|_p})$ given by

$$\mathrm{dom}_{t|_p} = \{u \mid pu \in \mathrm{dom}_t\}$$
$$\mathrm{val}_{t|_p}(u) = \mathrm{val}_t(pu)$$

Add example.

1 Deterministic Tree Automata

We will now define deterministic tree automata (DTA). Like a DFA, a DTA has finitely many transitions but it processes a tree. Intuitively, a DTA starts with some states at the leaves depending on the label of the leaves. At each step, based on the states of the children and the label of a node, the transition of the DTA determines the state of the node. When finally, the state of the root is determined, the input is accepted if this state is an accepting state; otherwise it is rejected. Let us define this automaton and its computation precisely.

Definition 4. A *deterministic tree automaton (DTA)* on Σ-labeled n-ary trees is $M = (Q, \Sigma, \delta, F)$ where Q is a finite set of states, $F \subseteq Q$ is a set of final/accepting states, and $\delta = \cup_{i=0}^{n} \delta_i$ is the transition function, where $\delta_i : Q^i \times \Sigma \to Q$.

We will now define the execution of a DTA M on a Σ-labeled tree $t = (\mathrm{dom}_t, \mathrm{val}_t)$. To do that let us recall some terminology. Recall that as per our convention of tree domains, a vertex $u \in \mathrm{dom}_t$ has i children if $uj \in \mathrm{dom}_t$ for all $j < i$ but $ui \notin \mathrm{dom}_t$. In particular, a vertex u has 0 children or is a leaf, if $u0 \notin domt$. We are now ready to define a run of DTA, an accepting run of a DTA, and the language recognized by a DTA.

Definition 5. The *run* of a DTA $M = (Q, \Sigma, \delta = \cup_{i=0}^{n}, F)$ on a tree $t = (\mathrm{dom}_t, \mathrm{val}_t)$ is a Q-labeled tree $\rho = (\mathrm{dom}_\rho, \mathrm{val}_\rho)$ where $\mathrm{dom}_\rho = \mathrm{dom}_t$ and for any vertex $u \in \mathrm{dom}_t$ with i children,

$$\mathrm{val}_\rho(u) = \delta_i(\mathrm{val}_\rho(u0), \ldots \mathrm{val}_\rho(u(i-1)), \mathrm{val}_t(u)).$$

A run $\rho = (\mathrm{dom}_\rho, \mathrm{val}_\rho)$ of M on t is *accepting* if $\mathrm{val}_\rho(\varepsilon) \in F$. A tree t is *accepted* by M if M has an accepting run on t. Finally the language recognized by M is the set of all Σ-labeled n-ary trees it accepts, i.e.,

$$\mathbf{L}(M) = \{t \mid M \text{ accepts } t\}.$$

Let us look at some examples to understand DTAs.

Example 6. Let Σ be the set $\{0, 1, \neg, \wedge, \vee\}$. Consider the DTA $M_{\mathsf{p}} = (\{q_0, q_1, q_{\mathsf{r}}\}, \Sigma, \delta, \{q_1\})$ where the transition function is given as

$$\delta_0(0) = q_0 \qquad \delta_0(1) = q_1$$
$$\delta_1(q_0, \neg) = q_1 \qquad \delta_1(q_1, \neg) = q_0$$
$$\delta_2(q_i, q_j, \wedge) = \begin{cases} q_1 & \text{if } q_i = q_j = q_1 \\ q_0 & \text{if } \{q_i, q_j\} \cap \{q_{\mathsf{r}}\} = \emptyset \\ & \text{and } \{q_i, q_j\} \cap \{q_0\} \neq \emptyset \end{cases} \qquad \delta_2(q_i, q_j, \vee) = \begin{cases} q_0 & \text{if } q_i = q_j = q_0 \\ q_1 & \text{if } \{q_i, q_j\} \cap \{q_{\mathsf{r}}\} = \emptyset \\ & \text{and } \{q_i, q_j\} \cap \{q_1\} \neq \emptyset \end{cases}$$

For any of the cases not considered above M_{p} transitions to the state q_{r}.

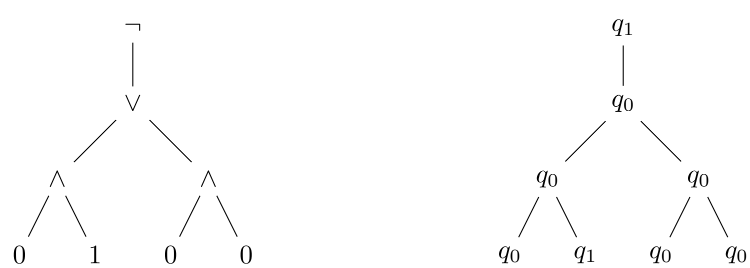

Figure 2: (a) Input tree $\neg((0 \wedge 1) \vee (0 \wedge 0))$ (left); (b) Run of DTA M_{p} on input $\neg((0 \wedge 1) \vee (0 \wedge 0))$ (right).

Recall that the string $\neg((0 \wedge 1) \vee (0 \wedge 0))$ represents the tree shown in Figure 2a. The run of M_{p} is shown in Figure 2b. Since the label of the root in the run is q_1, this input is accepted by M_{p}.

Trees over the alphabet Σ can be thought of as boolean expressions, provided the the labeling of the tree is consistent with the arity of the logical operators. If a tree represents a syntactically incorrect expression, like say a vertex labeled $\neg$ is either leaf or has two children, then M_{p} on such an input has a run where the state labeling the root is q_{r}. If the input tree corresponds to a syntactically correct boolean expression, then the state labeling the root in the run of M_{p} is q_0 is the boolean expression is false, and is q_1 if it is true. Therefore, the language recognized by M_{p} is the set of syntactically correct boolean expressions that evaluate to true.

Example 7. Consider the alphabet $\Sigma = \{0, 1, +, \cdot\}$. Consider the DTA $M_{\mathsf{a}} = (\{q_0, q_1, q_2, q_{\mathsf{r}}\}, \Sigma, \delta, \{q_0\})$ where

$$\delta_0(0) = q_0 \qquad \delta_0(1) = q_1$$
$$\delta_2(q_i, q_j, +) = q_{i+j \bmod 3} \text{ if } \{q_i, q_j\} \cap \{q_{\mathsf{r}}\} = \emptyset \quad \delta_2(q_i, q_j, \cdot) = q_{i \cdot j \bmod 3} \text{ if } \{q_i, q_j\} \cap \{q_{\mathsf{r}}\} = \emptyset$$

In all other cases not considred above, δ returns q_{r}.

Trees over Σ represent arithmetic expressions. If they are syntactically incorrect, then M_{a} will have a run whose root is labeled q_{r}. If the input is a syntactically correct arithmetic expression, M_{a}'s run will have root labels q_i, where i is the remainder when the value of the expression is divided by 3. Since the final state is q_0, the set of trees accepted by M_{a} are all those representing syntactically correct arithmetic expressions that evaluate to a value that is a multiple of 3.

Let us look at some examples of designing DTAs to recognize languages of trees.

Example 8. Consider the alphabet $\Sigma = \{0, 1, g, f\}$. The collection of all trees containing an even number of fs can be recognized by a DTA as follows. The DTA will track the parity of the number of fs seen in the subtree read so far. So the DTA has two states q_0 and q_1, and it will be in state q_i in subtree at position p is the number of fs in the subtree at position p modulo 2 is i. This can be ensured by the following transition rules.

$$\begin{array}{ll}
\delta_0(0) = \delta_0(1) = \delta_0(g) = q_0 & \delta_0(f) = q_1 \\
\delta_1(q_i, 0) = \delta_1(q_i, 1) = \delta_1(q_i, g) = q_i & \delta_1(q_i, f) = q_{i+1 \bmod 2} \\
\delta_2(q_i, q_j, 0) = \delta_2(q_i, q_j, 1) = \delta_2(q_i, q_j, g) = q_{i+j \bmod 2} & \delta_2(q_i, q_j, f) = q_{i+j+1 \bmod 2}
\end{array}$$

Since we need to accept trees with even fs, the set of final states will be $\{q_0\}$.

Example 9. Consider the context-free grammar with rules $S \to AA$, $A \to a \,|\, c \,|\, AB$, and $B \to b$. The set of derivation trees/parse trees of this grammar can be recognized by a DTA. Intuitively, the DTA's state after reading a subtree will track if the subtree is consistent with the grammar rules, and the state will record the nonterminal labeling the root of the subtree.

Formally, the states are $Q = \{q_a, q_b, q_c, q_A, q_B, q_C, q_*\}$, with $F = \{q_S\}$. Here the state q_* is reached if the subtree is not consistent with the rules of the grammar, and it is one of the other states q_X if it is consistent and the root of the subtree is labeled X. The transitions are as follows.

$$\begin{array}{lll}
\delta_0(a) = q_a & \delta_0(b) = q_b & \delta_0(c) = q_c \\
\delta_1(q_a, A) = q_A & \delta_1(q_c, A) = q_A & \delta_1(q_b, B) = q_B \\
\delta_1(q_A, q_B, A) = q_A & \delta_2(q_A, q_A, S) = q_S &
\end{array}$$

All "other" transitions go to q_*.

2 Regularity and Nonregularity

As in the case of words, these automata define an interesting class of tree languages that we will call *regular*.

Definition 10. A set A of Σ-labeled n-ary trees is a *regular* if there is a DTA M such that $A = \mathbf{L}(M)$.

Example 9 identifies an important class of tree regular languages — parse trees of a context-free grammar.

Theorem 11. *The set of derivation trees/parse trees of a context-free grammar $G = (N, T, P, S)$ is regular.*

Proof. The proof generalizes the construction in Example 9 and is based on the same intuition. Take $\Sigma = N \cup T$. The automaton recognizing the parse trees is as follows

- $Q = T \cup N \cup \{*\}$
- $F = \{S\}$
- $\delta_0(a) = a$ for $a \in T$, and for $i > 0$, $\delta_i(\alpha_1, \alpha_2, \ldots, \alpha_i, X) = X$ if $X \to \alpha_1\alpha_2 \cdots \alpha_i \in P$. In all other cases, $\delta_i(\alpha_1, \alpha_2, \ldots, \alpha_i, X) = *$.

□

The regularity of parse trees of a context-free grammar is an interesting connection between context-free languages and regular tree languages. However, Theorem 11 does not provide a characterization of regular tree languages, i.e., it is not the case that all regular tree languages are collections of parse trees of some grammar. Consider the collection B of all full binary trees labeled by alphabet $\Sigma = \{b, A\}$, with the property that (a) all leaves are labeled b and all internal vertices are labeled A, and (b) every path has at most 2 As and there is at least one path with exactly 2 As. The language B is, in fact, a finite set of trees that we can write as the following set of terms.

$$B = \{A(A(b, b), b), A(b, A(b, b)), A(A(b, b), A(b, b))\}.$$

B can be easily recognized by the DTA with states $\{q_0, q_1, q_2, q_*\}$ with final states $\{q_2\}$ and transition

$$\delta_0(b) = q_0 \quad \delta_2(q_i, q_j, A) = \begin{cases} q_{\max(i,j)+1} & \text{if } \max(i,j) < 2 \\ q_* & \text{otherwise} \end{cases}$$

with the assumption that δ returns q_* in all other cases. However, B is definitely not the set of all parse trees of a context-free grammar because if a grammar has a parse tree with two As then it can be pumped to create a tree with arbitrary number of As.

Like in the case of words, every finite set of trees is regular. We leave the proof as an exercise for the reader.

Proposition 12. *If A is a finite set of Σ-labeled trees then A is regular.*

Not all tree languages are regular; we could easily argue this through a counting argument as the number of DTAs is countable but the number tree languages is uncountable. However, we can give explicit examples of languages that are not regular. Such examples help understand the computational limits of DTAs.

Theorem 13. *There are tree languages that are not regular.*

Proof. Let $\Sigma = \{b, A\}$. Let $T(\Sigma)$ be the collection of all full binary trees where leaves are labeled by b and internal vertices are labeled by A. Consider

$$L = \{A(t,t) \mid t \in T(\Sigma)\}$$

That is, L consists of tree whose root is labeled by A, and its left and right subtrees are identical.

We will prove the non-regularity of L through a proof by contradiction. Assume that L is recognized by DTA M. Since M has finitely many states, there must be two trees t_1, t_2 with $t_1 \neq t_2$ such that the runs of M on t_1 and T_2 end in the same state. That is, if $\rho_1 = (\mathrm{dom}_{\rho_1}, \mathrm{val}_{\rho_1})$ is the run of M on t_1, and $\rho_2 = (\mathrm{dom}_{\rho_2}, \mathrm{val}_{\rho_2})$ is the run of M on t_2, then $\mathrm{val}_{\rho_1}(\varepsilon) = \mathrm{val}_{\rho_2}(\varepsilon)$. Since M accepts $A(t_1,t_1)$ and $A(t_2,t_2)$, it also accepts $A(t_1,t_2)$. But $A(t_1,t_2) \notin L$ which contradicts our assumption that M recognizes L. $\square$

3 Nondeterminism and Closure Properties

As for any machine model, we can consider a nondeterministic version of tree automata. The definition is identical to that for DTAs, except that the transition function returns a set of possible states for a vertex in the tree, given its label and the states of its children. Formally it can be defined as follows.

Definition 14. A *nondeterministic tree automaton (NTA)* on Σ-labeled n-ary trees is $M = (Q, \Sigma, \delta, F)$ where Q is a finite set of states, $F \subseteq Q$ is a set of final/accepting states, and $\delta = \cup_{i=0}^{n} \delta_i$ is the transition function, where $\delta_i : Q^i \times \Sigma \to 2^Q$.

Runs, acceptance and language recognized are natural generalization of Definition 5

Definition 15. The *run* of a NTA $M = (Q, \Sigma, \delta = \cup_{i=0}^{n}, F)$ on a tree $t = (\mathrm{dom}_t, \mathrm{val}_t)$ is a Q-labeled tree $\rho = (\mathrm{dom}_\rho, \mathrm{val}_\rho)$ where $\mathrm{dom}_\rho = \mathrm{dom}_t$ and for any vertex $u \in \mathrm{dom}_t$ with i children,

$$\mathrm{val}_\rho(u) \in \delta_i(\mathrm{val}_\rho(u0), \ldots \mathrm{val}_\rho(u(i-1)), \mathrm{val}_t(u)).$$

A run $\rho = (\mathrm{dom}_\rho, \mathrm{val}_\rho)$ of M on t is *accepting* if $\mathrm{val}_\rho(\varepsilon) \in F$. A tree t is *accepted* by M if M has an accepting run on t. Finally the language recognized by M is the set of all Σ-labeled n-ary trees it accepts, i.e.,

$$\mathbf{L}(M) = \{t \mid M \text{ accepts } t\}.$$

Like in the case of finite automata, nondeterminism does not add anything to the expressive power of tree automata.

Theorem 16. *Let A be a tree language recognized by an NTA. Then A is regular.*

Proof. The standard subset construction extends to this case. Let $N = (Q, \Sigma, \delta, F)$ be an NTA recognizing A. Consider the DTA $D = (2^Q, \Sigma, \delta', F')$ where

- $F' = \{P \subseteq Q \mid P \cap F \neq \emptyset\}$
- δ' is as follows: $\delta'_0 = \delta_0$ and $\delta'_k(Q_1, Q_2, \ldots, Q_k, f) = \{q | \exists q_1, q_2, \ldots, q_k.\ q_i \in Q_i \text{ and } q \in \delta_k(q_1, q_2, \ldots, q_k, f)\}$.

One can prove inductively the following observation for any Σ-labeled tree t. Let $\rho = (\mathrm{dom}_\rho, \mathrm{val}_\rho)$ be the run of D on t with $\mathrm{val}_\rho(\varepsilon) = P$. We can show that

$$q \in P \text{ iff there is a run } \rho' = (\mathrm{dom}_{\rho'}, \mathrm{val}_{\rho'}) \text{ of } N \text{ such that } \mathrm{val}_{\rho'}(\varepsilon) = q.$$

Using this observation we have $\mathbf{L}(D) = \mathbf{L}(N)$, given the definition of F'. ☐

Regular tree language enjoy similar closure properties to regular languages — they are closed un Boolean operations.

Theorem 17. *Regular tree languages are closed under all boolean operations.*

Proof. Standard constructions for DFAs like cross product and flipping final/non-final states, extend to DTAs. Details are left to the reader to work out. ☐

The notion of projection can be extended to trees. Let Σ and Γ be finite alphabets such that $|\Gamma| \leq |\Sigma|$. A projection is any function between such alphabets, i.e., $\pi : \Sigma \to \Gamma$. Given such a projection function, one can extend it to function from Σ-labeled trees to Γ-labeled trees in the natural way: for Σ-labeled tree $t = (\mathrm{dom}_t, \mathrm{val}_t)$ $\pi(t)$ is the Γ-labeled tree given by $(\mathrm{dom}_{\pi(t)}, \mathrm{val}_{\pi(t)})$ where $\mathrm{dom}_{\pi(t)} = \mathrm{dom}_t$ and $\mathrm{val}_{\pi(t)}(u) = \pi(\mathrm{val}_t(u))$. Extending projection to languages we have $\pi(A) = \{\pi(t) \,|\, t \in A\}$. Like regular (word) languages, regular tree languages are closed under projections.

Theorem 18. *If A is a regular tree language and π is a projection, then $\pi(A)$ is also regular.*

Proof. The construction of an NTA recognizing $\pi(A)$ is very similar to the way one constructs an NFA to recognize the projection of a word language. The details are left to the reader to work out. ☐

4 Decision Problems and MSO

We begin by looking at the classical decision problems for automata, namely, emptiness and universality. These problems are decidable for tree regular languages.

Theorem 19. *Given a NTA $M = (Q, \Sigma, \delta, F)$ there is a polynomial time algorithm to determine if $\mathbf{L}(M) = \emptyset$.*

Proof. Let us say a state $q \in Q$ reached on input t if there is a run $\rho = (\mathrm{dom}_\rho, \mathrm{val}_\rho)$ of M on t such that $\mathrm{val}_\rho(\varepsilon) = q$, i.e., q is reached on t if in some run of M on t, the label of the root is q. The idea is to inductively compute the set of all states that can be reached on some input. If this set can be computed then emptiness can be determined because *langM* is empty iff no final state is reachable. Let us define

$$E_i = \{q \mid \exists t \text{ of height at most } i \text{ and } q \text{ is reached on } t\}.$$

We will compute E_i inductively on i.

Observe that $E_0 = \{q \mid \exists a \in \Sigma.\ q \in \delta_0(a)\}$. Inductively,

$$E_i = E_{i-1} \cup \bigcup_k \{q \mid \exists a \in \Sigma. \exists q_0, \ldots q_{k-1} \in E_{i-1}.\ q \in \delta_k(q_0, q_1, \ldots q_{k-1}, a)\}.$$

Since M has finitely many states, for some ℓ (at most $|Q|$), $E_\ell = E_{\ell+1} = \{q | \exists t.\ q \text{ is reached on } t\}$. Therefore, $\mathbf{L}(M) = \emptyset$ iff $E_\ell \cap F = \emptyset$. ☐

Since regular tree languages are closed under complementation, it follows that checking universality is also decidable.

Corollary 20. *Given a DTA M, there is a polynomial time algorithm to check if $\mathbf{L}(M)$ contains all Σ-labeled trees.*

Proof. The result follows from Theorem 19 and Theorem 17. $\square$

Trees, like words, can be seen as structures. n-ary Σ-labeled trees are structures over the signature $\tau_T = \{<, \{S_i\}_{i=0}^{n-1}, \{Q_a\}_{a\in\Sigma}\}$. Here $<$ is interpreted as the ancestor/descendant relation, S_i the parent/child relationship, and Q_a the labeling function on the vertices of the tree. Formally, the correspondence between a tree and its representation as a τ_T structure is as follows.

Definition 21. A Σ-labeled n-ary tree $t = (\mathrm{dom}_t, \mathrm{val}_t)$ is the structure $\mathcal{T} = (T, <^{\mathcal{T}}, \{S_i^{\mathcal{T}}\}_{i=0}^{n-1}, \{Q_a^{\mathcal{T}}\}_{a\in\Sigma})$ where

- $T = \mathrm{dom}_t$
- $<^{\mathcal{T}}$ is the prefix relation on strings, i.e., $(u, v) \in <^{\mathcal{T}}$ if u is a prefix of v.
- For any i, $S_i^{\mathcal{T}} = \{(u, ui) \mid u, ui \in \mathrm{dom}_t\}$, and
- $Q_a^{\mathcal{T}} = \{u \mid \mathrm{val}_t(u) = a\}$.

Again there is a bijection between the structure representation of a tree t and the tree itself. Therefore, we will confuse the distinction between these representations.

Having identified the signature for labeled trees, and the structure representation of trees, first order logic or monadic second order logic *over trees* is just the logic over the restricted signature and when the models we consider are restricted to tree structures. We will say that a collection A of Σ-labeled n-ary trees is *definable* in first-order logic/MSO if there is a first-order/MSO sentence φ such that

$$A = \{t \mid t \models \varphi\}.$$

Analogous to the Büchi-Elgot-Trakhtenbrot theorem and Büchi's theorem, regularity and MSO definability are equivalent over tree structures.

Theorem 22 (Doner,Thatcher-Wright 1968)**.** *A tree language A is regular if and only if A is MSO definable.*

Proof. The proof is similar to Büchi's theorem for infinite words, or the Büchi-Elgot-Trakhtenbrot for finite words. To show regularity implies MSO definability, we construct a sentence that is satisfied by trees that are accepted by the tree automaton recognizing the regular language. Essentially the sentence says "there is a tree labeled by states of the automaton that is a valid accepting run of the machine on the tree structure." Conversely, we show that for any MSO formula φ we can construct a tree automaton that accepts trees that encode assignments and tree structures that satisfy φ. Assignments to variables and set variables can be encoded as extra binary labels on the tree, as in the case of words, and the automaton can be constructed inductively based on the structure of the formula φ; here the fact that regular tree languages are closed under boolean operators and projections is exploited. $\square$

5 Applications

We discuss a couple of applications of the theory of tree regular languages that we have introduced. The first is solving 2-player games, and the second is deciding various properties of rewriting systems.

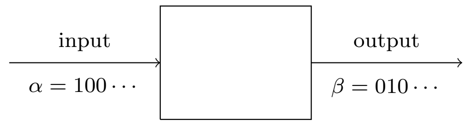

Figure 3: Schematic view of Church's problem.

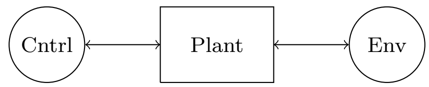

Figure 4: Setup of the Controller Synthesis Problem.

5.1 Solving Games

Alonzo Church was looking at the problem of circuit design. In a circuit design problem we might have an input/output specification, that dictates the sequence of outputs that must be produced for each sequence of inputs; this is shown schematically in Figure 3. The assumption here is that the input and output are a stream of bits. The input/output specification could be provided in many different ways, including possibly a logical formula that identifies the relationship between input and output. The goal is to construct a circuit that implements the specification. This problem, of *synthesizing* a circuit, to meet a given logical specification can be viewed as a game between two players — environment or Alice, and the system or Bob. The environment/Alice produces some sequence of inputs, and the system/Bob must respond to the input by constructing the output. Bob wins the game, if he can ensure that the output meets the specification. Now, Bob has a strategy to play this game in such a way that no matter what Alice does, he can win, then the strategy used by Bob is the circuit we are looking for.

A slight generalization of Church's problem is the *controller synthesis* problem, shown in Figure 4. Here there is a system/plant that is interacting with an environment. The goal is to synthesize a controller or an algorithm for the plant that ensures that the plant meets its specification no matter what the environment does. Again this can be viewed as a game between two player, namely the environment and the controller for the plant.

Both the controller synthesis problem and Church's synthesis problem can be solved automatically by developing algorithms that determine the winner in two-player games. It turns out that there is a close connection between solving such games and tree regular languages. We begin by formally defining the class of two-player games that we will study.

Definition 23. A *game graph* or *arena* is a $G = (Q_A, Q_B, E)$, where Q_A and Q_B are finite disjoint sets of vertices, and $E \subseteq (G_A \cup G_B) \times (G_A \cup G_B)$ is the edge relation with the property that evert vertex has at least one outgoing edge.

A two person game between Alice and Bob is formally a pair (G, φ), where G is a game graph and φ identifies the plays that are *winning for* Bob. The game is played as follows. Initially the "token" is placed in some vertex. In each step, the token is moved along an edge from the current vertex by one of the players. The player making the move is determined by whether the vertex containing the token belongs to Q_A or Q_B. If the token is in a vertex in Q_A, Alice makes the move, otherwise Bob makes the move.

Thus, a *play* from $q \in Q_A \cup Q_B$ is a sequence $q_1, q_2, \ldots$ where $q_1 = q$ and for every i, $(q_i, q_{i+1}) \in E$. There are different *winning conditions* one can consider.

Reachability For $G = (Q_A, Q_B, E)$, the *reachability* winning condition is specified by a set $F \subseteq Q_A \cup Q_B$. A play $\rho = q_1, q_2, \ldots q_i, \ldots$ is *winning for Bob* if for some i, $q_i \in F$.

Safety For $G = (Q_A, Q_B, E)$, the *safety* winning condition is specified by a set $F \subseteq Q_A \cup Q_B$. A play $\rho = q_1, q_2, \ldots q_i, \ldots$ is *winning for Bob* if for every i, $q_i \in F$.

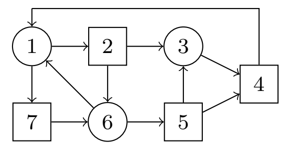

Figure 5: Game graph with positions belonging to Alice as squares.

Other There are many other types of games that are specified on infinite plays like recurrence, parity, etc.

In this section we will focus on reachability winning conditions.

Example 24. Consider a game graph shown in Figure 5. The vertices are parititioned into two sets — vertices in Q_A are shown as squares, while vertices in Q_B are shown as circles. Consider a reachability winning condition given by the set $F = \{3\}$. The play $1, 2, 6, 5, 3, \ldots$ is winning for Bob according to this condition. On the other hand, the play $(1, 7, 6)^\omega$ is not winning for Bob.

Notice that Bob is guaranteed to win if the game starts at vertex 3 no matter what Alice does. If the game starts from any other vertex, then Alice can play in manner that avoids visiting 3. This is because the only way to reach 3 is via vertices 2 and 5 which are positions in which Alice makes a move. Both vertices 2 and 5 have an outgoing edge that is not to 3, and so Alice can always avoid visiting 3.

Players in this game play it according to a *strategy*, which informs players on which edge to choose when it is their turn based on the history of the game so far. Identifying *winning strategies* then allows us to determine whether a player can win no matter the other player does. Let us define these notions precisely.

Definition 25. Consider a game graph $G = (Q_A, Q_B, E)$, with $Q = Q_A \cup Q_B$. A *strategy for Bob* from $q \in Q$ is a function $f : qQ^*Q_B \to Q$ specifying for any (finite) play prefix starting at q and ending in a vertex of Q_B, the edge that Bob should choose.

A play $\rho = q_1, q_2, \ldots, q_i, \ldots$ is *played according to strategy* f for Bob if for each $q_j \in Q_B$, $q_{j+1} = f(q_1 q_2 \cdots q_j)$.

A strategy from q is *winning for Bob* if every play from q played according to f is winning for Bob. A position q is *winning* for Bob, if Bob has a winning strategy from q. Finally the *winning region* for Bob is the set $W_B = \{q \mid q \text{ is winnign for Bob}\}$

Definitions of strategy for Alice, play according to a strategy for Alice, winning strategy for Alice, winning position for Alice, and winning region for Alice can all be defined analogously. We will denote the winning region for Alice by the set W_A. It is easy to see that a position cannot be winning for both Alice and Bob — if a position q is winning for Alice that means that Alice has a winning strategy no matter how Bob plays; and so Bob cannot win from q if Alice plays optimally. This gives us the following proposition.

Proposition 26. *The winning regions of Alice and Bob are disjoint.*

A game is said to be *determined* if every position is either winning for Alice or for Bob. That is, $W_A \cup W_B$ is the set of all vertices of the game graph. Reachability games are determined. They are a special class of a more general class of two-player games that were proved to be determined by Martin. We will not present its proof.

Theorem 27 (Martin 1975)**.** *Reachability games are determined.*

The computational problem associated with two-player games is the following: Given a game graph $G = (Q_A, Q_B, E)$ with a reachability winning condition F, and a position q, determine if q is winning for Bob. If Bob has a winning strategy from q, we may also want to construct the strategy. Being able to construct the strategy means that we can synthesize the controller/circuit in the synthesis problems that motivated these games. It turns out that these computational problems are indeed efficiently solvable. One way to prove these results is through a connection to regular tree languages.

Winning strategies for Bob from q in a reachability game can be thought of as a $Q_A \cup Q_B$-labeled finite trees where

- The root is labeled q;
- A vertex labeled $q_A \in Q_A$, has a child corresponding to each q' such that $(q_a, q') \in E$, and these are the only children;
- A vertex labeled $q_B \in Q_B$ has exactly one child, and if the child is labeled q' then $(q_b, q') \in E$;
- Every leaf is labeled by some vertex in F.

The label of the unique successor of a Q_B-vertex is the move that the strategy recommends for Bob. Notice that a play according to this strategy is simply a path in this tree, and since each path ends in a vertex in F, it is winning for Bob. We can now establish the connection between solving the game problem and regular tree languages.

Theorem 28. *The collection of winning strategies for Bob from position q is a regular tree language. The tree automaton recognizing this language can be effectively constructed.*

Proof. Given a $(Q_A \cup Q_B)$-labeled tree, the automaton needs to check that it satisfies all the conditions laid out in the previous paragraph. It is easy to see that they can be accomplished using finite memory.

The automaton's state at a vertex will either be the game position labeling the vertex or will be an error state — it will move to an error state if the subtree rooted at the vertex is not consistent with being a winning strategy. The formal construction of the automaton is as follows. Let $\overline{Q} = \{\overline{q} \mid q \in Q_A \cup Q_B\}$; thus, $\overline{Q}$ is just a copy of the game positions. The automaton is $M = (\overline{Q} \cup \{*\}, Q_A \cup Q_B, \delta, \{q\})$ where δ is defined as follows.

$$\delta_0(q) = \overline{q} \text{ if } q \in F \qquad \delta_1(\overline{q_1}, q) = \overline{q} \text{ if } q \in Q_B \text{ and } (q, q_1) \in E$$
$$\delta_i(\overline{q_1}, \overline{q_2}, \ldots \overline{q_i}, q) = \overline{q} \text{ if } q \in Q_B \text{ and } (q, q_1), \ldots (q, q_i) \text{ are the only outgoing edges of } q$$

In all other cases, δ returns $*$.

Notice the above DTA is linear in the size of the game graph. □

Theorem 29. *It can be decided whether Bob has a winning strategy from position q. Moreover if there is a winning strategy, it can be effectively constructed.*

Proof. One can check the emptiness of the language associated with the DTA recognizing winning strategies from Theorem 28; the emptiness checking algorithm can also produce a tree in the language and hence a winning strategy. The running time of this algorithm is linear. □

5.2 Rewrite Theories

Term rewriting is a general way to describe computation. States of the computation process are *terms*, and each step of the computation *rewrites* the term according to some rules. Thus, we can imagine such a rewrite system to be describing a directed graph (possibly infinite) where the vertices are terms and the edges are given by the rewriting rules. This approach to describing computation endows the state with some structure, and allows us to describe an infinite graph in a simple computationally effective manner. Rewrite systems are general enough to describe any computational model you might have encountered, like finite automata, context-free grammars, pushdown automata, and Turing machines.

Terms are nothing but labeled trees and rewrite rules will describe how trees can be transformed. Let us assume a finite set of labels Σ, and let us fix n to be the number of children any vertex in a tree. We will denote the set of Σ-labeled (n-ary) trees by $\mathcal{T}_\Sigma$. We now define a *(ground) rewriting system.*

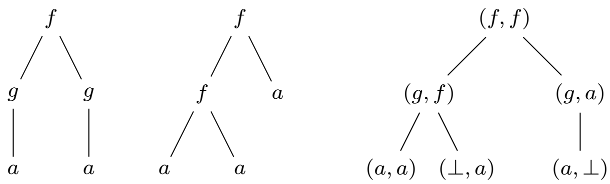

Figure 6: Encoding pairs of trees as a single tree. The Rightmost tree encodes the (ordered) pair for the first two trees.

Definition 30. A (ground) rewriting systems on n-ary Σ-labeled trees is a set of rules R where elements of R are rules of the form $\ell \to r$ with $\ell, r \in \mathcal{T}_\Sigma$.

Let us consider a Σ-labeled tree t. A rule of the form $\ell \to r$ can be applied to t, if ℓ is a subtree of t, and the result of applying the rule is to replace the subtree ℓ by r. This is called a rewriting step, and it can be applied repeatedly. Let us formally define this notion of rewriting.

Recall that for a tree $t = (\mathrm{dom}_t, \mathrm{val}_t)$, the subtree rooted at position p is denoted by $t|_p$. Let us define the notion of replacing on subtree by another. Recall also that for k trees $t_0, \ldots t_{k-1}$ and a symbol $f \in Sigma$, $f(t_0, \ldots t_{k-1})$ denotes the tree whose root is labeled f, and has as children $t_0, \ldots t_{k-1}$ (from left to right).

Definition 31. For trees $t = (\mathrm{dom}_t, \mathrm{val}_t)$ and u, and $p \in \mathrm{dom}_t$, the result of *replacing the subtree at position* p in t by u, is denoted by $t[u]_p$. Formally, it is inductively defined as follows.

- $t[u]_\epsilon = u$
- $f(t_0, t_1, \ldots, t_{k-1})[u]_{i.p} = f(t_0, \ldots, t_{i-1}, t_i[u]_p, t_{i+1}, \ldots t_{k-1})$

For example, when $t = f(a, f(a, g(b, f(a, b))))$, $t[b]_1 = f(a, b)$. Having defined the effect of replacing on esubtree by another, we can define a rewriting step according to a rule.

Definition 32. Given a set of rules R, *t reduces to u in one step* (with respect to R), denoted by $t \rightsquigarrow u$, if there is some $\ell \to r \in R$ and $p \in \mathrm{dom}_t$ such that $t|_p = \ell$ and $u = t[r]_p$.

t reduces to u zero or more steps (denoted by $t \overset{*}{\rightsquigarrow} u$) if there are $t_1, t_2, \ldots t_m$ such that $t_1 = t$, $t_m = u$ and $t_i \rightsquigarrow t_{i+1}$.

t reduces to u in one parallel rewrite step if there are postions $p_1, p_2, \ldots p_k \in \mathrm{dom}_t$ and rules $\ell_1 \to r_1, \ldots, \ell_k \to r_k$ such that $t|_{p_i} = \ell_i$ and u is the result of *simultaneously* replacing the subtrees $t|_{p_i}$ by r_i. We will denote this by $t \rightsquigarrow_{||} u$

Let us consider some examples to understand these definitions.

Example 33. Consider $\Sigma = \{a, b, g, h\}$. Let the set of rules be $g(b) \to h(b)$, and $h(g(b)) \to g(b)$. Consider the tree given by the term $t = f(a, g(b), g(b))$. Then $t \rightsquigarrow f(a, h(b), g(b))$, and $t \rightsquigarrow f(a, g(b), h(b))$. We could also say $t \overset{*}{\rightsquigarrow} f(a, h(b), h(b))$, as we could apply the rule $g(b) \to h(b)$ twice to get $f(a, h(b), h(b))$.

Now let us consider the tree $t' = f(a, h(g(b)), g(b))$. Some example rerwiting steps are $t' \rightsquigarrow f(a, g(b), g(b))$, $t' rewrtstr f(a, h(h(b)), h(b))$, and $t' rewrtstr f(a, h(b), h(b))$. We could also say $t' \rightsquigarrow_{||} f(a, g(b), g(b))$, $t' \rightsquigarrow_{||} f(a, g(b), h(b))$ and $t' \rightsquigarrow_{||} f(a, h(h(b)), h(b))$. However, t' does not rewrite to $f(a, h(b), h(b))$ in a single parallel rewrite step.

One of the results we would like to establish is the regularity of these different rewriting steps we have introduced in Definition 32. To do that, we need to identify how we can excode a binary relation on trees as a set of trees. The basis of this approach is to identify how a pair of trees can be seens as a single labeled tree; once this can be done, then the relation is simply the collection of all trees that encode pairs that belong to the relation. One way to encode a pair of labeled trees as a single labeled tree is shown in Figure 6. Here the component trees are "overlapped" to yield a tree over the product alphabet. One can formally define this "overlapping process as follows.

Definition 34. Let $t_1 = (\mathrm{dom}_{t_1}, \mathrm{val}_{t_1})$ and $t_2 = (\mathrm{dom}_{t_2}, \mathrm{val}_{t_2})$ be a pair of n-ary Σ-labeled trees. Let $\Sigma' = (\Sigma \cup \{\bot\}) \times (\Sigma \cup \{\bot\})$ be such that $\bot \notin \Sigma$. The encoding of the pair (t_1, t_2) is a Σ'-labeled n-ary tree $\mathrm{enc}(t_1, t_2) = t = (\mathrm{dom}_t, \mathrm{val}_t)$, where

- $\mathrm{dom}_t = \mathrm{dom}_{t_1} \cup \mathrm{dom}_{t_2}$, and
- $\mathrm{val}_t(u) = (\mathrm{val}_{t_1}(u), \mathrm{val}_{t_2}(u))$; here we assume that $\mathrm{val}_{t_i}(u) = \bot$, when $u \notin \mathrm{dom}_{t_i}$ for $i \in \{1, 2\}$.

The encoding of pairs of trees (Definition 34) can obviously also be generalized to k-tuples of trees. Having encoded pairs of trees, we can encode a binary relation as a tree language.

Definition 35. A binary relation $R \subseteq \mathcal{T}_\Sigma \times \mathcal{T}_\Sigma$ on n-ary Σ-labeled trees, can be encoded as a language of trees over the label set $\Sigma' = (\Sigma \cup \{\bot\}) \times (\Sigma \cup \{\bot\})$ as

$$\mathrm{enc}(R) = \{\mathrm{enc}(t_1, t_2) \mid (t_1, t_2) \in R\}$$

The relations $\rightsquigarrow$, $rewrtstr$, and $\rightsquigarrow_{||}$ are regular; this is a celebrated result due to Dauchet and Tison.

Lemma 36 (Dauchet-Tison)**.** *Given a set of rewrite rules R, the one step rewriting relation $\rightsquigarrow$, the many step rewriting relation $rewrtstrr$, and the parallel rewriting step $\rightsquigarrow_{||}$, are all regular tree languages under the encoding described in Definition 35.*

Proof. Showing that $\rightsquigarrow$ and $\rightsquigarrow_{||}$ are regular are easy, and is left as exercise. Showing $\overset{*}{\rightsquigarrow}$ is regular is beyond the scope of this course. ☐

The regularity of the reriwte steps (Lemma 36) can be exploited to get decidability results for ground rewriting systems. Consider the first order signature $\tau_{\mathsf{rewrite}} = (\mathcal{T}_\Sigma, \rightsquigarrow, rewrtstr, \rightsquigarrow_{||})$, where $\mathcal{T}_\Sigma$ are constant symbols. Thus, τ_{rewrite} is an infinite signature as it has infinitely many constant symbols. A set of rewrite rules R defines a structure over this signature, where $\rightsquigarrow, rewrtstr, \rightsquigarrow_{||}$ are intepreted to be the single rewrite step, the multiple rewrite step, and the parallel rewrite step with respect to the rules R.

Theorem 37 (Dauchet-Tison)**.** *Given a set of rewrite rules R, the set of first order sentences true in structure defined by R over the signature $\tau_{rewrite}$ is decidable.*

Proof. Like in the Doner-Thatcher-Wright theorem (and the Büchi-Elgot-Trakhtenbrot theorem), the validity of the theory of rewriting can be reduced to the emptiness question of tree automata, based on the closure properties of regular tree languages. Decidability follows fromthe decidability of emptiness problem for tree automata. ☐

Theorem 37 can be exploited to get a series of decidability results for different properties.

Pushdown Systems. Pushdown systems use information stored in finitely many states and an unbounded stack while computing (they are basically pushdown automata, without language accepting features, like final states). They are used to model recursive programs with variables over finite domains.

Corollary 38. *The first order theory of pushdown systems is decidable.*

Proof. A pushdown system can be modelled as a ground rewrite system. ☐

Normal Forms. u is said to be a *normal form* (with respect to rules R) if there is a no v such that $u \rightsquigarrow v$. u is said to be *normal form of* t if u is a normal form and $t \overset{*}{\rightsquigarrow} u$.

Corollary 39. *Checking if every t has a normal form is decidable.*

Proof. Existence of normal forms for all t can be expressed in first order logic.

$$\forall t \exists u((t \overset{*}{\rightsquigarrow} u) \wedge \neg(\exists v. u \rightsquigarrow v))$$

☐

Confluence. A set of rewrite rules R is said to be *confluent* if for every t, u, v such that $t \overset{*}{\rightsquigarrow} u$ and $t \overset{*}{\rightsquigarrow} v$ then there is a w such that $u \overset{*}{\rightsquigarrow} w$ and $v \overset{*}{\rightsquigarrow} w$. Informally, it means that the order in which rewrite rules are applied is not important (for some properties).

Corollary 40. *Checking if the rewrite system is confluent is decidable.*

Proof. Confluence can be expressed in first order logic

$$\forall t \forall u \forall v (((t \overset{*}{\rightsquigarrow} u) \wedge (t \overset{*}{\rightsquigarrow} v)) \rightarrow \exists w ((u \overset{*}{\rightsquigarrow} w) \wedge (v \overset{*}{\rightsquigarrow} w))) \qquad \square$$