

Tree Automata

Neel Krishnaswami

Department of Computer Science
Carnegie Mellon University

Why Tree Automata?

- Type declarations with refinement types declare a lattice of subtypes of an ML datatype.
- The \vee and \wedge operators in the type algebra create new entries in the lattice.
- Once we get such a type, we need to get the least upper bound of the type in the declared subtype lattice to figure out which refinement it is.
- Tree Automata and regular tree grammars describe sets of trees, and have well known (if exponential worst-case) algorithms for testing inclusion.

What Will Be Discussed

- Tree automata
- Tree grammars
- Implementation

Preliminaries: Ranked Alphabets

A *ranked alphabet* \mathcal{F} consists of an alphabet Σ and a function $a : \Sigma \rightarrow \mathbf{N}$. Examples are $x, y, nil, sin/1, cons/2$.

Arity-0 terms are written without the explicit arity marker. \mathcal{F}_i refers to $\{x \in \mathcal{F} \mid a(x) = i\}$.

Preliminaries: Trees

Assume the existence of a set of constants \mathcal{X} such that \mathcal{F} and \mathcal{X} are disjoint. Then define the set of trees $\mathbf{T}(\mathcal{F}, \mathcal{X})$ as the least set defined by:

- $\mathcal{F}_0 \subseteq \mathbf{T}(\mathcal{F}, \mathcal{X})$
- $\mathcal{X} \subseteq \mathbf{T}(\mathcal{F}, \mathcal{X})$
- $f(t_1, \dots, t_n) \in \mathbf{T}(\mathcal{F}, \mathcal{X})$ if $n \geq 1, t_1 \dots t_n \in \mathbf{T}(\mathcal{F}, \mathcal{X})$, and $f \in \mathcal{F}_n$

Preliminaries: Ground Terms and Substitutions

- A term t is *ground* if no $x \in \mathcal{X}$ appear in it, and the set of all ground terms $\mathbf{T}(\mathcal{F}, \emptyset)$ is written $\mathbf{T}(\mathcal{F})$.
- A term with variables is called *linear* if each variable appears at most once.
- A *substitution* is a mapping $\sigma : \mathcal{X} \rightarrow \mathbf{T}(\mathcal{F})$, with a finite domain.
- Let \mathcal{X}_n be a set of n variables. A *context* is a linear term $C \in \mathbf{T}(\mathcal{F}, \mathcal{X}_n)$. We write $C[t_1, \dots, t_n]$ to mean $\sigma(C)$, where $\sigma = \{t_1 \leftarrow x_1, \dots, t_n \leftarrow x_n\}$, and $\mathcal{C}(\mathcal{F})$ for all contexts over \mathcal{F} .

Bottom-up Tree Automata, Defined

A *tree automaton* is a tuple $A = (Q, \mathcal{F}, Q_f, \Delta)$, where Q is a set of states, the final states $Q_f \subseteq Q$, and Δ is a relation whose elements are written like this: $f(q_1, \dots, q_n) \rightarrow q$, with $n = a(f)$.

The Transition Relation

The transition relation $t \rightarrow_A t'$ is defined by:

$$t \rightarrow_A t' \Leftrightarrow \left\{ \begin{array}{l} \exists C \in \mathcal{C}(\mathcal{F} \cup Q), \\ \exists f(q_1, \dots, q_n) \rightarrow q \in \Delta, \\ t = C[f(q_1, \dots, q_n)], \\ t' = C[q] \end{array} \right\}$$

Bottom-up Tree Automata, Example

Let $\mathcal{F} = \{0, 1, \text{not}/1, \text{and}/2, \text{or}/2\}$, $Q = \{q_0, q_1\}$, and $Q_f = \{q_1\}$, and

$$\Delta = \left\{ \begin{array}{lll} 0 \rightarrow q_0 & \text{and}(q_0, q_0) \rightarrow q_0 & \text{or}(q_0, q_0) \rightarrow q_0 \\ 1 \rightarrow q_1 & \text{and}(q_1, q_0) \rightarrow q_0 & \text{or}(q_1, q_0) \rightarrow q_1 \\ \text{not}(q_1) \rightarrow q_0 & \text{and}(q_0, q_1) \rightarrow q_0 & \text{or}(q_0, q_1) \rightarrow q_1 \\ \text{not}(q_0) \rightarrow q & \text{and}(q_1, q_1) \rightarrow q_1 & \text{or}(q_1, q_1) \rightarrow q_1 \end{array} \right\}$$

Do an example evaluation of $\text{and}(\text{not}(0), \text{or}(0, \text{and}(1, 1)))$ on the board.

Various Properties

Many properties of bottom-up regular tree automata have analogues with string automata:

- Nondeterministic automata can be made deterministic with a subset construction, just like string automata.
- There's a pumping lemma, only it works on the height of trees, rather than the length of the string.
- Deterministic automata can be minimized, using a version of the Myhill-Nerode theorem generalized to trees.

Top-down Tree Automata

Nondeterministic top down tree automata are just like bottom-up tree automata, except that the final set Q_f gets replaced with an initial set Q_i , and the transitions change to the form:

$$q(f(x_1, \dots, x_n)) \rightarrow f(q_1(x_1), \dots, q_n(x_n))$$

Deterministic Top Down Automata Are Weaker

However, deterministic top-down automata recognize a smaller set of languages. For example, consider $L = \{f(a, b), f(a', b')\}$. If you try to construct a top-down deterministic automaton, you run into the problem that the rule for the function symbol f is of the form $q(f(x_1, x_2)) \rightarrow f(q_1(x_1), q_2(x_2))$. So the state q_1 must recognize both a and a' , and likewise for q_2 . So the smallest language L' a top-down deterministic automaton containing L must be $\{f(a, b), f(a', b), f(a, b'), f(a', b')\}$.

This is what leads to the *tuple distributivity* restriction in the Yardeni and Shapiro paper.

Closure properties

Recognizable tree languages are closed under *complementation*, *union*, and *intersection*. I won't offer the proofs, but will describe the constructions used.

To get the complement language, start with a deterministic automaton for it, and then swap the final and non-final states. This is possibly exponential if you start with an NFTA, as usual.

Union and intersection can be found using *product automata*

Product Automata for Union and Intersection

Suppose you have two automata $A_1 = (Q_1, \mathcal{F}, Q_{f1}, \Delta_1)$ and $A_2 = (Q_2, \mathcal{F}, Q_{f2}, \Delta_2)$, which recognize languages $L(A_1)$ and $L(A_2)$. To construct a product automaton, let:

$$Q' = Q_1 \times Q_2$$
$$\Delta' = \left\{ f((q_1^1, q_1^2), \dots, (q_n^1, q_n^2)) \rightarrow (q_1, q_2) \mid \begin{array}{l} f(q_1^1, \dots, q_n^1) \rightarrow q^1 \in \Delta_1, \\ f(q_1^2, \dots, q_n^2) \rightarrow q^2 \in \Delta_2 \end{array} \right\}$$

For union, let $Q'_f = (Q_{1f} \times Q_2) \cup (Q_1 \times Q_{2f})$, and for intersection let $Q'_f = Q_{1f} \times Q_{2f}$.

Complexity of Decision Problems

- Testing membership is linear in the size of the term t .
- Testing the emptiness of the language an automaton A recognizes takes time proportional to $|A|$. (The trick is to compute the set of reachable states and then to see if any of them are in the final set.)

Complexity of Decision Problems

- The finiteness of the accepted language can be checked in time proportional to $|A|$, by checking for loops in the transition rules.
- An inclusion algorithm is easy to get since we have closure under intersection, union and complement, and an emptiness test. Unfortunately, this is potentially exponential time for NFTAs.

Regular Tree Grammars

A *regular tree grammar* $G = (S, N, \mathcal{F}, R)$ is given by an *axiom* S , a set of *non-terminals* N (with $S \in N$), and a set of production rules $A \rightarrow \alpha$, where $A \in N$ and $\alpha \in \mathbf{T}(\mathcal{F} \cup N)$.

Example:

$$\begin{aligned} \textit{List} &\rightarrow \textit{nil} \\ \textit{List} &\rightarrow \textit{cons}(\textit{Nat}, \textit{List}) \\ \textit{Nat} &\rightarrow 0 \\ \textit{Nat} &\rightarrow \textit{succ}(\textit{Nat}) \end{aligned}$$

Normalized Tree Grammars

A *normalized* regular tree grammar is a tree grammar in which all of the production rules are of the form $A \rightarrow f(A_1, \dots, A_n)$ or $A \rightarrow a$, where the A_i are nonterminals and the f, a are elements of \mathcal{F} . Every regular tree grammar can be converted into a normalized tree grammar, by first discarding all the non-productive and non-reachable nonterminals, and then introducing new nonterminals for nested productions.

Equivalence of Automata and Tree Grammars

Given a normalized grammar $G = (S, N, \mathcal{F}, R)$, we can construct a tree automaton to recognize it as follows:

$$Q = \{Q_A \mid A \in N\}$$

$$Q_f = Q_S$$

$$f(q_{A_1}, \dots, q_{A_n}) \rightarrow q_A \in \Delta \text{ if and only if } A \rightarrow f(A_1, \dots, A_n) \in R$$

Likewise for the other direction.

Algorithms for Fast Automata Operations

The main tricks that Aiken and Murphy use in their implementation are:

- Maintain the system in a form close to a normalized tree grammar (or an NFTA), rather than in a deterministic form, in order to avoid potential exponential space growth.
- Maintaining the rules in a form that enables fast emptiness tests.
- Maintaining the rules in a form that enables “fast” negation.

Representation Tricks: Leaf-linear form

Aiken and Murphy maintain their regular tree language in a form they call *leaf-linear form*. This is essentially a variation of a normalized regular tree grammar. First, they add two special terminal symbols – 0 and 1. 0 matches no terms, and 1 matches all ground terms. (Their syntax also apparently permits conjunctions, but that's an illusion.)

Fast emptiness tests

Aiken and Murphy maintain the invariant that whenever a particular production rule generates the empty set, it shall be represented with the 0 symbol. This makes testing emptiness an $O(1)$ operation, though they have maintain the invariant (which they do by simplifying expressions until they can be sure they are 0).

Taking the union of two leaf-linear systems that maintain the emptiness invariant is trivial; simply add the new equations $S_{new} = S_1$ and $S_{new} = S_2$. Maintaining the emptiness requirement is a trivial check.

Fast emptiness tests: Intersection

Intersection is a lot harder; its worst-case time is exponential, since there is no choice but to use the product construction. However, Aiken and Murphy found it possible to do better in the average case. They took a system of equations with axioms S_1 and S_2 , and then defined $S' = S \cup \{x = \text{Simp}(S_1 \wedge S_2)\}$, and then simplified the set of equations. At the last step, they identified all of the terms $x_i \wedge x_j$ and introduced new variables for them.

Fast emptiness tests: Negation

Aiken and Murphy report that tree negation does suffer from exponential blowup in practice. They deal with this problem by using a conservative approximation to negation (whether this is a subset or superset depends on the problem for which the answer will be used). After propagating the negation for some number of steps k through the set of equations, they simply returned either 0 or 1, depending.

Question: Do refinement types ever generate negations? It *looked* like they don't, but I didn't fully understand Frank's talk last week.

Inclusion

I am not sure their inclusion test results are relevant to refinement types. They are trying to solve a very different decision problem from the one that arises in refinement type-checking. Aiken and Murphy want to solve $\forall\sigma.\sigma(S_1) \subseteq \sigma(S_2)$ and $\exists\sigma.\sigma(S_1) \subseteq \sigma(S_2)$, where S_1 and S_2 can contain variables.

Other optimizations

The main other optimization they used was simply memoizing any of the tree operations, and maintaining the memos in one hash table for each operation.

Correction

The relationship between context-free grammars and tree automata is more complicated than I suggested: for any tree automaton, the *fringe* of its trees can be recognized by a context-free grammar, and vice versa.

The set of trees themselves is bigger; for example, they can be used to decide temporal logic propositions.

Algebraic Definition of Tree Automata

An \mathcal{F} -algebra is a pair $\mathcal{A} = (Q, \alpha)$, where Q is a set (called the *carrier* of \mathcal{A}) and α is a function that assigns to each $f \in \mathcal{F}$ a function $\alpha_f : Q^{\text{arity}(f)} \rightarrow Q$. (\mathcal{F} is the usual ranked alphabet.)

Homomorphisms on \mathcal{F} -algebras

Suppose that $\sigma : \mathcal{X} \rightarrow Q$ is an *assignment* of variables to elements of the carrier. This determines a unique homomorphism h_σ from $\mathbf{T}(\mathcal{F}, \mathcal{X})$ into Q :

- $h_\sigma(x \in \mathcal{X}) = \sigma(x)$
- $h_\sigma(a \in \mathcal{F}_0) = \alpha_a$
- $h_\sigma(f(t_1, \dots, t_n)) = \alpha_f(h_\sigma(t_1), \dots, h_\sigma(t_n))$ where $f \in \mathcal{F}_n$

Clearly, all h_σ are the same on all ground terms; we write $h_{\mathcal{A}}$ to restrict the domain to ground terms.

Tree Automata Defined

A tree automaton A can be seen as a pair (\mathcal{A}, Q_f) , where \mathcal{A} is an \mathcal{F} -algebra and $Q_f \subseteq Q$, and $|Q|$ is finite.

A ground term t is *accepted* if $h_{\mathcal{A}} \in Q_f$.

Algebra and Substitution

The big win to thinking in algebraic terms is that we can easily change the carrier from “states” to any other set. For example, take Q to be a finite subset of $\mathbf{T}(\mathcal{F}, \mathcal{X})$, and the unique homomorphism on the algebra defines substitution for us.

Linear homomorphisms*

A *linear homomorphism* g is a mapping for each $f \in \mathcal{F}$ to a linear term $t_f \in \mathbf{T}(\mathcal{F}', \mathcal{X}_{arity(f)})$, extended to trees such that:

- $g(a) = t_a$ for $a \in \mathcal{F}_0$
- $g(f(t_1, \dots, t_n)) = t_f\{x_1 \leftarrow g(t_1), \dots, x_n \leftarrow g(t_n)\}$

* These are called *pure finite-state transforms* in the Thatcher paper I got the proof from.

Proof of closure under linear homomorphisms

Automata languages are closed under linear homomorphisms. That is, given an automaton $A = (\mathcal{A}, Q_f)$, and a linear homomorphism g , the image of the language recognized by A , $g(L(A))$ is also a recognizable language.

Sketch of Proof

Suppose we have an automaton $A = (Q, \mathcal{F}, Q_f, \Delta)$ which recognizes a language $L(A)$. We can construct an automaton to recognize $g(L(A))$ using the following construction $A' = (Q', \mathcal{F}', Q'_f, \Delta')$. We get Δ' using the following procedure: consider each rule $f(q_1, \dots, q_n) \rightarrow q \in \Delta$, and the linear term t_f . We define a set of states $Q^r = \{q_p^r | p \in Pos(t_f)\}$, and for all positions p :

$$\begin{aligned} t_f(p) = f' \in \mathcal{F}'_k &\Rightarrow f'(q_{p1}^r, \dots, q_{pk}^r) \rightarrow q_p^r \in \Delta_r \\ t_f(p) = x_i &\Rightarrow q_i \rightarrow q_p^r \in \Delta_r \\ q_\epsilon^r &\rightarrow q \in \Delta_r \end{aligned}$$

Sketch of Proof

- $Q' = Q \cup \bigcup_{r \in \Delta} Q^r$
- $Q'_f = Q_f$
- $\Delta' = \bigcup_{r \in \Delta} \Delta_r$

You can then do an induction on the reduction to get the desired result. This proof depends critically on the linearity assumption!