Umeå University
Department of Computing Science

Lecture notes on

Tree Automata

prepared by the participants of the course
Formal Languages
(spring term 09)

Frank Drewes (ed.)

Preface

These are the lecture notes of a slightly odd course on tree automata that was held by me during the spring term 2009. When the regular course this was supposed to be was cancelled in autumn 2008, a small group of interested students and I discussed whether something could be done, anyway. We agreed on a "light" version with a sparse schedule, consisting of lectures on selected topics that the students would compile lecture notes of. The first version of each chapter was reviewed by one of the fellow students and revised accordingly. After another revision that was coordinated by Petter Ericson, the notes were compiled into the current document. Altogether, it took quite a while, but eventually we finished writing it.

Be invited to read on if you want to get a first glimpse of
what tree automata are and what they may be good for!

However, be critical as well. This is not a book written by experts in the field, but a collection of lecture notes prepared by students who have not been exposed to that much formal language theory before. Moreover, the time the students could invest in it was less than what would normally have been the case. Consequently, not everything is perfect, and there are almost certainly mistakes that neither the reviewer nor I have discovered. That said, I think the text can be a valuable source of information especially for other students who, without struggling with too much formalism, want to find out whether this is a matter worth reading more about.

If you, after that, decide that you indeed would like to know more about tree automata, there are numerous sources. Of course, there are thousands of original research articles, but there are also books and general survey articles. Thus, you may have a look at them before going on and reading the original literature in specific areas of tree automata theory. The earliest book by Gécseg and Steinby summarizing the state of the art of the theory in the early 1980s is [GS84]; an updated version can be found in [GS97]. In the beginning of the 1990s, a collection of survey articles was edited by Nivat and Podelski [NP92]. A book by Fülöp and Vogler focussing on the use of tree transducers for syntax-directed semantics of programming languages is [FV98]. The most up-to-date collection of survey articles on tree automata, which is available for free on the Internet, is [CDG$^+$07]. Tree automata (in the form of tree grammars and tree transducers) also play a central role for so-called tree-based picture generation in [Dre06], using the idea explained in Section 5 of Chapter V. Finally, a very recent survey by Fülöp and Vogler on weighted tree automata (which are not covered by these lecture notes) is [FV09].

Umeå, 19 April 2010
Frank Drewes

Contents

Chapter I

Bottom-Up and Top-Down Tree Automata

by Adam Sernheim

The theory of tree languages and tree automata generalises the theory of string languages and ordinary finite automata. It started in the 1960s, when the bottom-up tree automaton and the equivalent regular tree grammar were introduced and studied by Brainerd, Doner, Mezei, Thatcher, and Wright. [Bra68, Bra69, Don65, Don70, MW67, Tha67, TW68]. To understand the idea behind the generalization, recall that a string is a sequence of symbols taken from some alphabet, see Figure 1.

Figure 1: String

A string can be turned 90° and be viewed as a special case of a tree in which every node has at most one child, called a monadic tree. By removing this restriction, general trees over a given alphabet are obtained; see Figure 2.

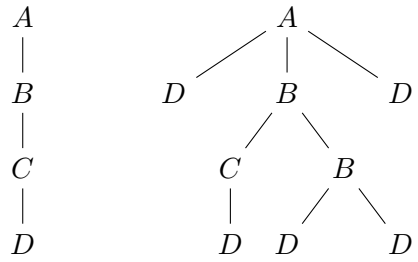


Figure 2: A monadic tree corresponding to the string in Figure 1 and a non-monadic tree

In the string case the task of a automaton is to check if a string belongs to a certain language. Analogously, in the case of trees, we want to know if a tree belongs to a certain set of trees, called a tree language.

1 Alphabets and Trees

The first step if we want to work with trees and tree automata is to define what a tree is. There are several examples of trees in both computer science and in mathematics. In computer science we encounter trees of many different kinds, for example binary search trees, parse trees, arithmetic expressions, structured documents such as XML-documents, and many more. The trees describing the structure of XML documents belong to a family of trees called unranked trees, these will be discussed in Chapter V. Since trees are an important special case of graphs (namely graphs in which there is exactly one path between each pair of nodes), trees and their algorithmic properties are extensively studied in graph theory. In these lecture notes, trees are mainly viewed as formal expressions that generalise strings in the sense of Figures 1 and 2 and are processed by some type of finite automaton. These trees are usually built using symbols from a ranked alphabet.

1.1 Ranked Alphabet

We mostly want to consider trees in which the label of each node determines how many children this node has. This is called the rank of the node in question. Because of this we define a *ranked alphabet*, where each symbol in the alphabet has a rank. With this alphabet we can construct ranked trees.

Definition 1 (Ranked alphabet) *A* ranked alphabet *is a tuple* (Σ, rk) *where*

- Σ *is a finite alphabet and*
- rk *is a mapping,* $rk\colon \Sigma \to \mathbb{N}$.

Definition 1 lets us create sets of symbols, each of which is given a specific rank. Σ is called alphabet to indicate that its elements are used as symbols. However it can be any finite set. The mapping rk gives each element $f \in \Sigma$ a unique rank, $rk(f)$. Several symbols can have the same rank.

Notations

- We write Σ for (Σ, rk).
- $\Sigma^{(k)} = \{f \in \Sigma \mid rk(f) = k\}$ is the set of symbols in Σ with rank k.
- For $f \in \Sigma$, writing $f^{(k)}$ indicates that $rk(f) = k$.

Example 2 The ranked alphabet $\Sigma = \{a^{(2)}, b^{(2)}, c^{(1)}, d^{(0)}, e^{(0)}\}$ describes an alphabet where a and b have rank two, $rk(a) = rk(b) = 2$, c have rank one, $rk(c) = 1$, and the symbols d and e have rank zero, $rk(d) = rk(e) = 0$.

Each symbol in Σ has a unique rank, according to the definition, but it can sometimes be convenient to disregard this restriction. Σ can then, for example, contain both $f^{(2)}$ and $f^{(1)}$. Formally, this can be achieved by providing the symbols with indices, $f_2^{(2)}$ and $f_1^{(1)}$, and using the convention that these indices are not explicitly written out.

1.2 Trees

We can now define the notion of trees used throughout the remainder of these lecture notes, with the exception of Chapter V.

Definition 3 (Tree) *A* (ranked) tree *over the ranked alphabet* Σ *is a string over the symbols in* $\Sigma \cup \{\text{`[', `]', `,'}\}$ *(where* `[', `]' *and* `,' *are not in* Σ*) of the form* $f[t_1, \ldots, t_k]$*, where*

- $f \in \Sigma^{(k)}$ *for some* $k \in \mathbb{N}$ *and*
- $t_1, \ldots, t_k$ *are trees over* Σ*.*

The symbol f is the root symbol of the tree and $t_1, \ldots, t_k$ are the subtrees. Symbols f of rank 0 result in leaves and make the recursion in the definition stop.

Notations

- We generally omit the brackets for the leaves. Thus, if $rk(f) = 0$, instead of writing $f[]$ we write f. Thus, $\Sigma^{(0)}$ can be viewed as a set of trees.
- T_Σ is the set of all trees over Σ.
- For a set T of trees, $\Sigma(T) = \{f[t_1, \ldots, t_k] \mid f^{(k)} \in \Sigma, t_1, \ldots, t_k \in T\}$.

The notation $\Sigma(T)$ is used when we do not want to work with the whole set of trees over Σ. An example is if we want the subtrees to be more specified, e.g. of maximum height 5 or contain at least one specific symbol. These notations will make it easier for us later on.

Example 4 The string $a[b[d, c[d]], a[e, e]]$ is a tree over the ranked alphabet in Example 2. The tree is shown in Figure 3. Thus, we draw trees in the usual manner, with the root on top and the children of each node underneath it, ordered from left to right.

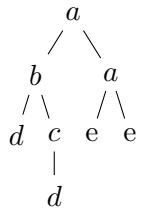


Figure 3: A tree over the ranked alphabet in Example 2

From the notation we can see that the tree belongs to $\Sigma(\Sigma(\{d, c[d], e\}))$.

2 Finite-State Tree Automata (FTA)

When defining the first type of tree automata considered in these lecture notes, we start from the definition of *Non-Deterministic Finite Automata* (NFA) in string language theory and expand it to the case of trees. The NFA reads a

symbol and moves to a set of states. It then reads another symbol and goes to another set of states, depending on the previous states. In the tree case we would read the symbols in the leaves and go to a group of states. In the next step the automaton would read a symbol and depending on the states from the children move to a new set of states. Strings can be read forwards or backwards. The direction in which a string is read does not affect the power of the automaton, although it can affect the size. In the tree case this is true for the non-deterministic case but not in the deterministic case; see the discussion in Section 3, where it is shown that deterministic top-down tree automata are much weaker. For this reason, we begin constructing the tree automaton to read from the leaves and upwards, bottom up.

Definition 5 (Bottom-up NFTA) *A* bottom-up non-deterministic finite-state tree automaton *(NFTA) is a tuple* $A = (Q, \Sigma, \delta, F)$ *where*

- Q *is a finite set of* states *viewed as symbols of rank* 0*,*
- Σ *is the ranked* input alphabet *where* $Q \cap \Sigma = \emptyset$*,*
- $\delta \colon \Sigma(Q) \to \wp(Q)$ *is the* transition function*, where* $\wp(Q)$ *denotes the power set of* Q*, and*
- $F \subseteq Q$ *is the set of* accepting states*.*

For a tree $t = f[t_1, \ldots, t_k])$*, we define*

$$\delta^*(t) = \bigcup \{\delta(f[q_1, \ldots, q_k]) \mid q_1 \in \delta^*(t_1), \ldots, q_k \in \delta^*(t_k)\}.$$

The language accepted by A *is* $L(A) = \{t \in T_\Sigma \mid \delta^*(t) \cap F \neq \emptyset\}$*.*

Notations

- We write $\delta(f, q_1, \ldots, q_k)$ instead of $\delta(f[q_1, \ldots, q_k])$ when using the transition function.
- We write $(q_0, .., q_k) \xrightarrow{f} q$ when the automaton goes to state q from states $q_0, \ldots, q_k$ upon reading the symbol f of rank k.

When δ^* is applied to a tree t, it recursively goes down to the leaves where δ is applied. The result from δ, when applied to the leaves is the set of states that the automaton is in initially on this lowest level of the tree. After that the automaton applies δ to the parent nodes of the leaves intuitively resulting in another set of states. In this way the automaton continues upwards to the root. Thus the result from δ^* is a set of states. If the intersection between the $\delta^*(t)$ and the set of accepting states is non-empty then the automaton accepts the tree as a member of the tree language.

Example 6 We have the alphabet $\Sigma = \{f^{(2)}, g^{(1)}, a^{(0)}, b^{(0)}\}$. We want to create a NFTA A over T_Σ that non-deterministically chooses the leftmost a and checks that there is no g on the path from the leftmost a to the root. $L(A) = \{t \in T_\Sigma \mid t$ contains at least one a and there is no g on the path from the leftmost a to the root$\}$

First create the transition function for the leftmost a,

$$\xrightarrow{a} q_l.$$

For all other a in the tree,

$$\xrightarrow{a} q.$$

The last transition function for leaves is the one that reads a b,

$$\xrightarrow{b} q_b.$$

One restriction on the NFTA is that it should not allow a g to be read on the path starting at the leftmost a. Transitions that read a g after the leftmost a are omitted because they are superfluous since the automaton will reject a tree with a g after the leftmost a. We are left with the following transitions for the symbols g and f:

$$q \xrightarrow{g} q, q_b \xrightarrow{g} q_b$$

$$(q_l, q) \xrightarrow{f} q_l, (q_b, q_l) \xrightarrow{f} q_l$$

$$(q, q) \xrightarrow{f} q, (q_b, q_b) \xrightarrow{f} q_b.$$

The set of accepting states for the NFTA is q_l.

Let us now turn to the deterministic case of a tree automaton. A tree automaton is deterministic if the transition function returns at most one state for each argument. Thus, formally,

$$|\delta(f, q_1, \ldots, q_k)| \leq 1$$

for all $f^{(k)} \in \Sigma$ and $q_1, \ldots, q_k \in Q$. Regardless of where in the tree we are, and which input symbol we read, there is at most one possible state to continue in. From this we get $|\delta^*(t)| \leq 1$ for all t. If there is no possible state to go to, the automaton rejects the tree. The deterministic finite string automata in the literature are often total as well. If this definition would be adopted, all transitions had to return exactly one state, $|\delta(f, q_1, \ldots, q_k)| = 1$.

In the case of string automata deterministic and non-deterministic are equally powerful, i.e. they can recognise the same class of languages. This is also the case for bottom-up tree automata.

Theorem 7 *For every bottom up NFTA, A, there exists a total bottom-up DFTA, A', such that $L(A) = L(A')$*

Proof For an NFTA $A = (Q, \Sigma, \delta, F)$, we define an equivalent DFTA $A' = (\wp(Q), \Sigma, \delta', F')$. Here, $F' = \{Q' \subseteq Q \mid Q' \cap F \neq \emptyset\}$ and $\delta'(f, Q_1, \ldots, Q_k) = \bigcup\{\delta(f, q_1, \ldots, q_k) \mid q_i \in Q; 1 \leq i \leq k\}$ for $f^{(k)} \in \Sigma$ and $Q_1, \ldots, Q_k \subseteq Q$. A' is by definition a DFTA.

$L(A) = L(A')$ holds if $\delta^*(t) = \delta'^*(t)$ for all $t \in T_\Sigma$, because of our choice of F'.

The proof is by induction over the structure of the tree, $t = f[t_1, \ldots t_k]$, as follows:

$$\begin{aligned}
\delta^*(t) &= \bigcup\{\delta(f, q_1, .., q_k) \mid q_i \in \delta^*(t_i)\} && \text{(by definition of } \delta^*) \\
&= \bigcup\{\delta'(f, q_1, \ldots, q_k) \mid q_i \in \delta'^*(t_i)\} && \text{(by induction hypothesis)} \\
&= \delta'(f, \delta'^*(t_1), \ldots, \delta'^*(t_k)) && \text{(by definition of } \delta') \\
&= \delta'^*(t) && \text{(by definition of } \delta'^*)
\end{aligned}$$

Note that the base case $k = 0$, in which there are no trees t_i, is included. This completes the proof. ∎

3 On the Top Down Case

A string can be read from both left and right. A string automaton that reads its input from the right is as powerful as one that reads from the left. We can read trees from the bottom up and from the top down, but these two modes do not produce equally powerful tree automata in the deterministic case. However, non-deterministic top-down tree automata are as powerful as bottom-up NFTA.

To define a top-down NFTA we consider an alternative representation of the calculations of bottom-up NFTA.

Consider a bottom-up NFTA $A = (Q, \Sigma, \delta, F)$ as in Definition 5. We define $\langle \Sigma, Q\rangle = \{\langle f, q\rangle^{(k)} \mid f^{(k)} \in \Sigma, q \in Q\}$, a ranked alphabet of symbol-state pairs, where the pairs have the same rank as the symbol.

A *computation by* A on a tree $t = f[t_1, \ldots t_k]$ ending in the state q is a tree $\langle f, q\rangle[t'_1, \ldots t'_k]$ in $T_{\langle \Sigma, Q\rangle}$ such that there exists a transition $(q_1, \ldots q_k) \xrightarrow{f} q$ in A and $t'_1, \ldots t'_k$ are computations on $t_1, \ldots t_k$ that end in $q_1, \ldots q_k$. There exists a computation on t that ends in q if and only if $q \in \delta^*(t)$. As a consequence, $t \in L(A)$ if and only if there exists a computation ending an accepting state.

We use this alternative definition of the semantics of bottom-up NFTA to define a corresponding top-down version.

Definition 8 (Top-down NFTA) *A* top-down NFTA *is a tuple* $A = (Q, \Sigma, \delta, S)$ *where*

- Q *and* Σ *are the same as in Definition 5, with the difference being that the states are now seen as symbols of rank* 1*,*
- δ *is a family of transition functions* $\delta_f \colon Q \to \wp(Q^k)$ *for* $f^{(k)} \in \Sigma$*, and*
- $S \in Q$ *denotes the initial state of the automaton.*

A computation on a tree t *starting in* q *is a tree* $\langle f, S\rangle[t'_1, \ldots t'_k]$ *so that* $t'_1, \ldots t'_k$ *are computations on* $t_1, \ldots t_k$ *starting in* $q_1, \ldots q_k$*, where* $(q_1, \ldots q_k) \in \delta_f(q)$*. A accepts* t *if it there exists a computation on* t *that starts in* S*.*

We need to define a deterministic top-down FTA in order to show that it does not have the same power as the bottom-up NFTA.

Definition 9 (Top-Down DFTA) *A top-down NFTA $A = (Q, \Sigma, \delta, S)$ is* deterministic *if $|\delta_f(q)| \leq 1$ for all $q \in Q$ and $f \in \Sigma$. In this case, the mapping $\tilde{\delta}\colon T_\Sigma \to \wp(Q)$ is defined recursively by*

$$\tilde{\delta}(f[t_1, \ldots, t_k] = \{q \mid \delta_f(q) \in \tilde{\delta}(t_1) \times \cdots \times \tilde{\delta}(t_k)\},$$

yielding the set of all states $q \in Q$ such that there exists a computation starting in q. Hence, $t \in L(A)$ if $S \in \tilde{\delta}(t)$.

By "reversing transitions", it is easy to show that top-down NFTAs recognise the same class of tree languages as bottom-up NFTAs and, thus, bottom-up DFTAs. We now prove that top-down DFTAs are not as powerful as these three classes, by showing that they do not even recognise all finite tree languages. (It is easy to show that every finite tree language L can be recognised by a bottom-up DFTA whose states correspond to the subtrees of the trees in L.)

Theorem 10 *There are tree languages that are recognisable by bottom-up DFTAs but not by any deterministic top-down FTA.*

Proof Let $\Sigma^{(0)} = \{a, b\}$ and $\Sigma^{(1)} = \{f\}$. Consider the finite tree language $L = \{f[a, b], f[b, a]\}$. Suppose that a top-down DTFA $A = (Q, \Sigma, \delta, q_0, S)$ recognises L. Let $\delta_f(S) = \{(q_1, q_2)\}$. Since $f[a, b] \in L(A)$, we must have $q_1 \in \tilde{\delta}(a)$ and $q_2 \in \tilde{\delta}(b)$. But, since $f[b, a] \in L(A)$, we also have $q_1 \in \tilde{\delta}(b)$ and $q_2 \in \tilde{\delta}(a)$. This means that $S \in \tilde{\delta}(f[a, a])$ (and $S \in \tilde{\delta}(f[b, b])$), which shows that the top-down DFTA A accepts $f[a, a]$. This contradicts the assumption that $L(A) = L$. ∎

Chapter II

Regular Tree Languages

by Johan Granberg

In this chapter we are going to look at and define regular tree languages. By definition, this is the class of tree languages that are generated by regular tree grammars [Bra69]. It turns out that this class is identical to the class of languages recognisable by NFTA and Bottom up NFTA and DFTA. We will look at how they relate to string languages and show that *regular* tree languages are closely related to *context-free* string languages.

1 Regular Tree Grammars

A regular tree grammar is a device that generates trees by applying productions that replace nonterminals of rank 0 with subtrees of nodes and nonterminals.

Definition 1 (Regular Tree Grammars) *A Regular Tree Grammar (RTG) is a tuple $G = \{N, \Sigma, P, S\}$, where*

- *N is a finite set of non-terminal symbols of rank 0,*
- *Σ is a finite ranked alphabet,*
- *P is a finite set of productions $A \to t$ with $A \in N$ and $t \in T_{\Sigma \cup N}$, and*
- *$S \in N$ is the initial nonterminal.*

A derivation step $s \to_p s'$, where $s, s' \in T_{\Sigma \cup N}$ is obtained by selecting an occurrence of a nonterminal A in s and a production $A \to t$ in P and constructing s' from s by replacing the selected nonterminal occurrence with t.
The language defined by G is denoted $L(G)$ and is given by

$$L(G) = \{s \in T_\Sigma | S \to_P^* s'\},$$

where $\to_P^*$ denotes the transitive and reflexive closure of $\to_P$.[1]

If it is clear from context the P in $\to_P$ can be omitted. The language does not change if the derivations are done in parallel, that is, instead of in every derivation step selecting a single nonterminal, a set of nonterminals in the tree

[1]The transitive reflexive closure $a \to^* b$ is defined as a sequence of applications over $a_1, \cdots, a_n$ where $n \geq 1$, $a = a_1$, $b = a_n$ and $a_1 \to a_2 \to \cdots \to a_n$.

can be selected and replaced using (potentially different) productions in P. This is because the order in which productions are applied to distinct nonterminals is of no importance — hence they can be applied in parallel.

2 Tree Grammars Producing Strings

Regular tree grammars are closely related to context free string grammars in (at least) two different ways:

- Regular tree grammars are special cases of context free string grammars as the right hand side of a rule is a string that can be interpreted as a tree and the left hand side is a nonterminal. (These strings are built using symbols from N and Σ and auxiliary brackets and commas.)
- Every generated tree can be seen as the parse tree of the string that is given by reading the leaves of the tree from left to right. This sequence of leaves is called the yield of a tree.

The yield can be computed by the yield function that is defined as follows.

Definition 2 (yield of a tree) *Let $t = f[t_1, ..., t_k]$ be a tree. The* yield *of t is the string $yield(t)$ defined recursively as follows:*

$$yield(t) = \begin{cases} f & if\, k = 0 \\ yield(t_1) \cdots yield(t_k) & if\, k > 0. \end{cases}$$

For a function $f\colon A \to B$ and a subset S of A, let $f(S) = \{f(a) \mid a \in S\}$. For example, $yield(L) = \{yield(t) \mid t \in L\}$ for every tree language L.

As mentioned above, the yield of a tree can be seen as the result of a parse tree. Thus, $yield(L)$ corresponds to the set of all strings having a correct parse tree (with respect to a given grammar). For the case of regular tree languages this yields the following formal result.

Theorem 3 *Let CFL and RTL denote the classes of context-free string languages (not containing the empty string) and regular tree languages, respectively. Then $yield(RTL) = CFL$.*

Proof *($\subseteq$)* Let $G = (N, \Sigma, R, S)$ be a regular tree grammar. Consider the context-free grammar $G' = (N, \Sigma_0, R', S)$ with $R' = \{A \to yield(t) \mid (A \to t) \in R\}$. It follows by a straightforward structural induction on t that, for every tree $t \in T_{\Sigma \cup N}$, $t \to_R t'$ implies $yield(t) \to_{R'} yield(t')$. Conversely, for all strings w such that $yield(t) \to_{R'} w$, there is a tree t' such that $t \to_R t'$ and $yield(t') = w$. By induction on the length of derivations, this yields $L(G') = yield(L(G))$.

($\supseteq$) Now let $G = (N, \Sigma, R, S)$ be a context-free grammar without empty right-hand sides. Let K be the set of all $|w|$ such that w is a right-hand side of a rule in R, and define $\Delta = \{\sigma^{(0)} \mid \sigma \in \Sigma\} \cup \{*^{(0)} \mid k \in K\}$. Consider the regular tree grammar $G' = (N, \Delta, R', S)$ such that, if $A \to a_1 \cdots a_k$ is in R, then $A \to *[a_1, \ldots, a_k]$ is in R'.

Clearly, if the construction of the first direction of the proof is applied to G', then G is obtained. Hence, $yield(L(G')) = L(G)$. ∎

3 Recognising Tree Languages

To recognise a tree language a tree automaton can be used. To construct an automaton that recognises a specific tree language $L(G)$ generated by a regular tree grammar G, one can proceed as follows:

- normalise the grammar.
- turn around the rules to obtain transitions (where nonterminals become states).

In the following, this is made more precise.

Definition 4 *A regular tree grammar $G = (N, \Sigma, P, S)$ is in* normal form *if all right-hand sides of rules in P are in $\Sigma(N)$. In other words, only the root symbol of the right-hand side is an output symbol, whereas all children are nonterminals.*

Lemma 5 *Every regular tree grammar G can be rewritten to a regular tree grammar in normal form G', such that $L(G') = L(G)$.*

Proof
Step 1: Let $P' = \{(A \to t) \in P \mid t \in N\}$ where $G = (N, \Sigma, P, S)$. Define $P_0 = \{A \to t' \mid (A \to t) \in P \setminus P' \text{ and } t \to^*_{P'} t'\}$. In P_0 we thus have no rules left whose right-hand side is in N. It should be clear that $G_0 = (N, \Sigma, P_0, S)$ generates the same language as G.
Step 2: As long as there exists a rule $A \to f[t_1, \cdots, t_k]$ and an index i, $1 \leq i \leq k$, such that $t_i \notin N$, add a new nonterminal A' and replace the rule by $A \to f[t_1, \cdots, t_{i-1}, A', t_{i+1}, \cdots, t_k]$ and $A' \to t_i$. Clearly, this does not affect $L(G)$. After a finite number of steps of this kind the grammar is in normal form. Note that termination is guaranteed since each of the right-hand sides $f[t_1, \cdots, t_{i-1}, A', t_{i+1}, \cdots, t_k]$ and t_i are strictly smaller than $f[t_1, \cdots, t_k]$. ∎

Once the grammar is rewritten to normal form a tree automaton can be be created by turning nonterminals into states and "turning rules around" to obtain the transitions of the automaton. More precisely, every rule $A \to f[A_1, \cdots, A_n]$ is turned into the transition $(A_1, \cdots, A_n) \xrightarrow{f} A$. It should be obvious that the automaton obtained in this way accepts exactly $L(G)$ if S is the (only) accepting state. Since the construction can obviously be reversed, we get the following result:

Theorem 6 *A tree language is recognisable by a bottom-up NFTA if and only if it is regular.*

4 Algebras

Algebras define the semantic interpretation of a tree similarly to how tree automata define which trees follows a syntax. By combining a tree automaton with an algebra we can define a language in an arbitrary domain $\mathbb{A}$. In this way, semantics is kept separate from syntax.

Definition 7 *Let Σ be a ranked alphabet.*
A Σ-algebra is a pair $\mathcal{A} = (\mathbb{A}, \mathcal{F})$, where

- *$\mathbb{A}$ is an arbitrary set called the* domain *of $\mathcal{A}$ and*
- *$\mathcal{F}$ is a family of functions $\mathcal{F} = (f_\mathcal{A})_{f \in \Sigma}$ with $f_\mathcal{A} \colon \mathbb{A}^k \to \mathbb{A}$ for $f^{(k)} \in \Sigma$.*

In particular, if $k = 0$ then $f_\mathcal{A} \colon \mathbb{A}^0 \to \mathbb{A}$, which corresponds to a constant in $\mathbb{A}$ represented by a function that takes no input and returns an element in $\mathbb{A}$.

Trees in T_Σ can be evaluated with respect to a Σ-algebra by means of the mapping $Val_\mathcal{A}(t) \colon T_\Sigma \to \mathbb{A}$, which is defined as follows:

$$Val_\mathcal{A}(t) = f_\mathcal{A}(\, Val_\mathcal{A}(t_1), ..., \, Val_\mathcal{A}(t_k))$$

for all trees $t = f[t_1, \cdots, t_k]$.

If $L \subseteq T_\Sigma$ is a tree language then $Val_\mathcal{A}(L) = \{\, Val_\mathcal{A}(t) \mid t \in L\}$ is a subset of $\mathbb{A}$. This means that by combining tree automata for syntax with appropriate algebras for semantics we can specify subsets of arbitrary domains. This division gives us a clear separation between the syntactic structures (the trees in L) and their semantics (the result of evaluating them in $\mathbb{A}$). The algebra and the automaton cooperate to express a language of elements of $\mathbb{A}$.

5 Combinations of Regular Tree Grammars and Algebras

As algebras can be used to assign semantics to any tree language there is a large number of uses for them. A few examples are given by the string algebra, the free term algebra and the YIELD algebra. Notationally calligraphic uppercase letters will be used to represent algebras and blackboard bold uppercase letters will be used to represent the domain of those algebras.

Definition 8 (String algebra) *The string algebra $\mathcal{S}$ (with respect to a given ranked alphabet Σ) has the domain $\mathbb{S} = (\Sigma^{(0)})^*$. Its operations are defined as follows:*

- *for $a^{(0)} \in \Sigma$, $a_\mathcal{S} = a$ and*
- *for $f^{(k)} \in \Sigma, k > 0$, $f_\mathcal{S}(u_1, ..., u_k) = u_1 \cdots u_k$.*

Given that, it should be clear that $Val_\mathcal{S}(t) = yield(t)$. This gives us another formulation of when a string language is context free, that is of Theorem 3.

Theorem 9 *A string language L is context free if and only if there is a regular tree language L' such that $L = Val_\mathcal{S}(L')$.*

Our next well-known example of algebra is the so-called free term algebra.

Definition 10 (Free term algebra) *The free term algebra $\mathcal{F}_\Sigma$ has the domain $\mathbb{T}_\Sigma = T_\Sigma$. For $f^{(k)} \in \Sigma$ and $t_1, \cdots, t_k \in T_\Sigma$: $f_{T_\Sigma}(t_1, ..., t_k) = f[t_1, ..., t_k]$.*

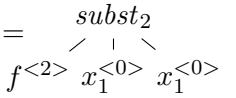
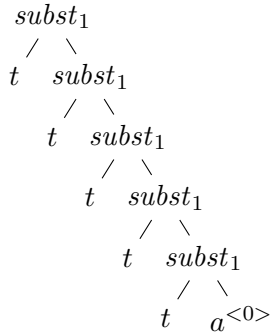
Given this definition, we obviously have $Val_{\mathcal{F}_\Sigma}(t) = t$ for all trees $t \in T_\Sigma$. In other words, $Val_{\mathcal{F}_\Sigma}$ is simply the identity. This shows us that nothing is necessarily lost when using an algebra to assign a semantics to the trees in some tree language.

Definition 11 (YIELD algebra) *The YIELD algebra $\mathcal{Y}$ over Σ and the variables in $X_n = \{x_1^{(0)}, ..., x_n^{(0)}\}$ (where $\Sigma \cap X_n = \emptyset$) has the domain $\mathbb{Y} = T_{\Sigma \cup X_n}$. The signature it is defined over contains the following symbols, which are defined together with their respective operations.*

- *For all $f^{(k)} \in \Sigma \cup X_n$, there is a symbol $f^{<k>}$ of rank 0, where $Val_\mathcal{Y}(f^{<k>}) = f_\mathcal{Y}^{<k>} = f[x_1, \cdots, x_k]$.*
- *For all $k \leq n$, there is a symbol $subst_k^{(k+1)}$. Given trees $t_0, \cdots, t_k \in T_{\Sigma \cup X_n}$, we define $subst_{k_\mathcal{Y}}(t_0, \cdots, t_k)$ to be the tree obtained from t_0 by replacing every occurrence of x_i (where $i \in \{1, ..., k\}$) by t_i.*

Thus, the semantic interpretation of $subst_k$ takes the first argument t_0 and replaces any variable x_i in it with the argument tree t_i.

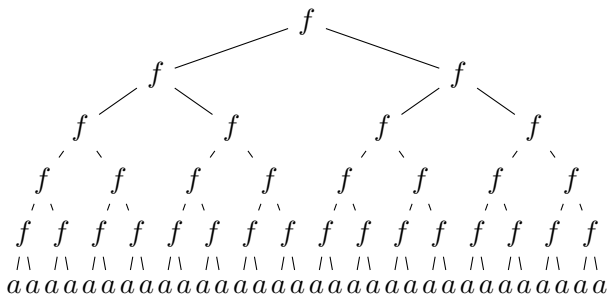
Example 12

Let $t =$  in the tree $t_5 =$ 

If we apply the YIELD algebra to this tree (which is mostly extending in one branch), the inner-most subtree $subst_1[t, a^{<0>}]$ evaluates to $f[a, a]$ by first evaluating t to $Val_\mathcal{Y}(t) = f[x_1, x_1]$ and then replacing the variables in $f[x_1, x_1]$ with the value of the second child of $subst_1$ (which is $Val_\mathcal{Y}(a^{<0>}) = a$).

Similarly, evaluating the second lowest $subst_1$, we replace the variables in t with the value $f[a, a]$ yielding $f[f[a, a], f[a, a]]$.

By continuing this evaluation all the way to the root we will create a tree looking like this



We started evaluating the initial tree bottom up by means of recursion as this evaluation order ensures that all child nodes are evaluated before being used as arguments to its parents corresponding *subst*-function.

Generalising the example a bit, we may consider the trees

$$t_n = subst_1[t, subst_1[t, \cdots subst_1[t, a^{<0>}] \cdots]]$$

that consist of n nested occurrences of $subst_1$ with t as the first parameter. Then $Val_{\mathcal{Y}}(t_n)$ is the fully balanced tree with 2^n leaves.

We can easily produce a regular tree grammar G such that $L(G) = \{t_n \mid n \in \mathbb{N}\}$. According to the observations made above, we have $yield(\, Val_{\mathcal{Y}}(L(G))) = a^{2^n}$, which is not context free. According to theorem 9 $Val_{\mathcal{Y}}(L(G))$ is thus not a regular tree language.

On the other hand it can easily be shown that every regular tree grammar G can be rewritten to a G' such that $Val_{\mathcal{Y}}(L(G')) = L(G)$. The combination of the YIELD algebra with regular tree grammars is thus strictly more powerful than only regular tree grammars.

Theorem 13 *The regular tree languages are a proper subset of the tree languages of the form $Val_{\mathcal{Y}}(L(G))$, where $\mathcal{Y}$ is a YIELD algebra and G is a regular tree grammar.*

In fact, it can be proved that the monadic tree languages of the form $Val_{\mathcal{Y}}(L(G))$ are precisely the context-free string languages (up to the details of representation; see the discussion at the beginning of Chapter I).

Chapter III

Tree Transformations and Transducers

by Lucas Lindström

In this chapter we are going to explore the concept of a tree transformation, which is a function that maps one tree to another in a possibly non-deterministic manner. To this end, we will define a computational model that realises this function, which we call a tree transducer. The most basic types of tree transducers were introduced soon after the first papers on tree automata and tree grammars. In particular, Rounds and Thatcher invented the top-down and the bottom-up tree transducer [Rou68, Rou70, Tha70, Tha73].

The first section of this chapter will focus on general notions and notations concerning tree transformation. The second section will define and describe different types of tree transducers. Some concepts that will be discussed are elaborated upon in [Eng75] and [Bak79].

1 Tree Transformations

A *tree transformation* is a binary relation $\tau \subseteq T_\Sigma \times T_{\Sigma'}$. We use $\tau(t)$ to denote the set $\{t' \in T_{\Sigma'} \mid (t, t') \in \tau\}$ for all $t \in T_\Sigma$. This can be thought of as a non-deterministic function, as the output is the set of all trees in $T_{\Sigma'}$ that t maps to. If $|\tau(t)| \leq 1$ for all $t \in T_\Sigma$ then τ is considered to be a (partial) function $\tau\colon T_\Sigma \to T_{\Sigma'}$.

Notations

- For $t \in T_{\Sigma \cup X_k}$ and $t_1, \ldots, t_k \in T_{\Sigma'}$, $t[\![t_1, \ldots, t_k]\!]$ denotes the tree obtained from t by replacing each occurrence of x_i, where $i \in \{1, \ldots, k\}$ by t_i. (Note that while t_i can contain variables, this replacement is not recursive.)
- For a set T of trees, $T_\Sigma(T)$ denotes the set of all trees $t[\![t_1, \ldots, t_k]\!]$ such that $t \in T_{\Sigma \cup X_k}$ and $t_1, \ldots, t_k \in T$, where $k \in \mathbb{N}$. Notice that $T_\Sigma(T)$ contains all of T_Σ since t need not contain any variables, which implies that $T_\Sigma(\emptyset) = T_\Sigma$. To mention another special case, t may consist of only a single variable, which means $T \subseteq T_\Sigma(T)$. In fact, as an inductive definition, $T_\Sigma(T)$ is the smallest set of trees such that:
  1. $T \subseteq T_\Sigma(T)$ and
  2. $\Sigma(T_\Sigma(T)) \subseteq T_\Sigma(T)$.

2 Tree Transducers

In this section we will define a couple of common types of tree transducers, go through some examples that describe their use, and look at how they relate to each other in terms of expressiveness. We start with the top-down tree transducer, which is one of the most useful classes of tree transducers.

Definition 1 (top-down tree transducer) *A* top-down tree transducer *(*td transducer *for short), is a tuple* $td = (\Sigma, \Sigma', Q, R, q_0)$*, where*

- Σ *and* Σ' *are the* input and output alphabets*, respectively,*
- Q *is a finite set of states which are symbols of rank 1,*
- R *is a finite set of rules of the form* $q[f[x_1, \ldots, x_k]] \to t[\![q_1[x_{i_1}], \ldots, q_l[x_{i_l}]]\!]$ *with* $q, q_1, \ldots, q_l \in Q, f \in \Sigma_k, t \in T_{\Sigma'}(X_l), i_1, \ldots, i_l \in \{1, \ldots, k\}$ *and* $x_{i_1}, \ldots, x_{i_l} \in X_k$ *(in other words,* $R \subseteq Q(\Sigma(X)) \times T_{\Sigma'}(Q(X))$*, where* $X = \{x_1, x_2, \ldots\}$*), and*
- $q_0 \in Q$ *is the* initial state*.*

A td transducer can be deterministic or non-deterministic. The difference between the types is that the deterministic case requires the left-hand side of each rule in R *to be unique to the set.*

For trees $s, s' \in T_{\Sigma'}(Q(T_\Sigma))$ there is a computation step $s \to_{td} s'$ if s can be written as $s = s_0[\![q[f[t_1, \ldots, t_k]]]\!]$ with $q \in Q$ (where s_0 contains exactly one occurrence of x_1), R contains a rule as above, and

$$s' = s_0[\![t[\![q_1[t_{i_1}], \ldots, q_l[t_{i_l}]]\!]]\!].$$

The tree transduction computed by td is given by $td(t) = \{t' \in T_{\Sigma'} \mid q_0[t] \to^*_{td} t'\}$ for all $t \in T_\Sigma$. [2]

Example 2 (deterministic case) Let $\Sigma = \{f^{(2)}, g^{(1)}, a^{(0)}\}$ and $\Sigma' = \{F^{(2)}, G^{(1),a^{(0)}}\}$, $Q = \{start, copy, mirror\}$ and $q_0 = start$. Let R consist of the following rules:

$$\begin{aligned}
start[f[x_1, x_2]] &\to F[start[x_1], start[x_2]] \\
start[g[x_1]] &\to G[F[copy[x_1], mirror[x_1]]] \\
start[a] &\to a \\
\\
copy[f[x_1, x_2]] &\to F[copy[x_1], copy[x_2]] \\
copy[g[x_1]] &\to G[copy[x_1]] \\
copy[a] &\to a \\
\\
mirror[f[x_1, x_2]] &\to F[mirror[x_2], mirror[x_1]] \\
mirror[g[x_1]] &\to G[mirror[x_1]] \\
mirror[a] &\to a
\end{aligned}$$

[2] The star superscript suggests any number of computation steps, i.e. $\to^*_{td}$ is the transitive and reflexive closure of $\to_{td}$; see also Definition 1.

Simply put, this transducer traverses the tree from the root and copies or mirrors some subtrees according to the rules stated above, depending on what input symbol it encounters. More precisely, starting at the root, the input tree is copied to the output (turning f into F and g into G) until the top-most g's are reached. For each of them, the second rule in R duplicates the subtree, using states *copy* and *mirror* to output the first copy unchanged and to mirror the second copy. (Note that the mirroring takes place in the last but third rule.)

Observation 3 *All computations of a td transducer will terminate, since the depth of the subtree of each state decreases for each step. The height of the output tree is linearly bounded from above in the height of the input tree, and the size of the output tree is exponentially bounded in the size of the input tree. This is because the height of the output tree can be no larger than c times the height of the input tree, where c is the largest height of t over all rules* $q[f[x_1, \ldots, x_k]] \to t[\![q_1[x_{i_1}], \ldots, q_l[x_{i_l}]]\!]$. *The bound on the size follows directly from this, together with the fact that the rank of symbols in the output trees is bounded.*

Now we will introduce a different type of tree transducer. The *bottom-up tree transducer* differs from the td transducer in that it begins its computation at the leaves of the tree and works its way towards the root.

Definition 4 (bottom-up tree transducer) *A* bottom-up tree transducer *(bu* transducer *for short), is a tuple* $bu = (\Sigma, \Sigma', Q, R, F)$, *where*

- R *is a finite set of rules* $f[q_1[x_1], \ldots, q_k[x_k]] \to q[t[\![x_{i_1}, \ldots, x_{i_l}]\!]]$ *where* $q, q_1, \ldots, q_k \in Q$, $f \in \Sigma_k$, $t \in T_{\Sigma'}(X_l)$ *and* $i_1, \ldots, i_l \in \{1, \ldots, k\}$, *and*
- $F \in Q$ *is a set of final states.*

A bu transducer can be deterministic or non-deterministic as in the case of a td transducer.

Intuitively, the bu transducer begins by assigning states to leaf nodes, and at the same time transmuting each node. Once a state has been assigned to each child of a node, that node is itself assigned a state, and is transmuted according to the transducer rule set. This process is repeated until the root node has been assigned a state, which is required to belong to the set of final states.

Example 5 (deterministic case) Let $\Sigma = \{f^{(2)}, g^{(1)}, a^{(0)}\}$, $\Sigma' = \{F^{(1)}, G^{(1)}, H^{(2)}, a^{(0)}\}$, $Q = \{p, q\}$ and $F = \{q\}$. Let R be defined by the following rules:

$$\begin{aligned} a &\to p[a] \\ f[p[x_1], p[x_2]] &\to p[F[x_1]] \\ g[p[x_1] &\to q[H[G[x_1], G[x_1]]] \\ f[q[x_1], q[x_2]] &\to q[H[x_1, x_2]] \end{aligned}$$

This bu transducer accepts those trees (in the sense that $bu(t) \neq \emptyset$) in which all paths from the leaves to the root contain exactly one g, and performs various transformations. The second subtree of f branches are removed according to

the second rule of R if no g has been encountered below it. Subtrees of g are duplicated under a new H branch if no previous g has been encountered below it according to the third rule. Finally, f branches where a g has been encountered in both children are turned into H branches according to the fourth rule.

Example 6 (non-deterministic case) The rule set of a non-deterministic bu transducer might look like this:

$$\begin{aligned} a &\to q[a] \\ f[q[x_1]] &\to q[f_1[x_1]] \\ f[g[x_1]] &\to q[f_2[x_1]] \\ g[q[x_1]] &\to q[g[x_1, x_1]] \end{aligned}$$

This transducer, if completed, would non-deterministically change f into either f_1 or f_2, and would also duplicate the subtrees of any g. a is left unchanged. And respectively, the rule set of a non-deterministic top-down transducer, which performs a somewhat similar operation:

$$\begin{aligned} q[a] &\to a \\ q[f[x_1]] &\to f_1[q[x_1]] \\ q[f[x_1]] &\to f_2[q[x_2]] \\ q[g[x_1]] &\to g[q[x_1], q[x_1]] \end{aligned}$$

The reader may wish to figure out why these two tree transducers to *not* compute the same tree transformation.

Clearly, Observation 3 is valid even for bottom-up tree transducers, using the same line of arguments. However, this does not mean that both formalisms are equivalent. In fact, there are tree transformations that can be realised with a td transducer but not with a bu transducer, and vice versa. In other words, the two devices are incomparable with respect to their transformational power, i.e., neither class of tree transformations is contained in the other.

Example 7 Let $\Sigma = \{a^{(1)}, e^{(0)}\}$ $\Sigma' = \{a^{(2)}, b^{(2)}, e^{(0)}\}$, and $Q = \{q\}$. Let R_{td} consist of the rules

$$\begin{aligned} q[a[x_1]] &\to a[q[x_1], q[x_1]] \\ q[a[x_1]] &\to b[q[x_1], q[x_1]] \\ q[e] &\to e \end{aligned}$$

and let R_{bu} consist of

$$\begin{aligned} a[q[x_1]] &\to q[a[x_1, x_1]] \\ a[q[x_1]] &\to q[b[x_1, x_1]] \\ e &\to q[e] \end{aligned}$$

This defines a td and a bu transducer, using R_{td} and R_{bu}, respectively. Since the td transducer can non-deterministically choose to produce either a or b as output when consuming each a from the input alphabet, the transducer can generate all balanced trees over Σ'. This is impossible for the bu transducer, because it can only copy the subtrees of the state in each step. On the other hand, for this very reason, the bu transducer can guarantee that all symbols on each level in the tree will be the same, which the td transducer cannot.

This limitation can be overcome by applying two transducers of the same type on the input tree in sequence, as is formally described in the next theorem.

Theorem 8 (See [Eng75], [Bak79]) *Let TD and BU denote the classes of top-down and bottom-up tree transductions respectively. Then the following holds:*

1. *$TD \not\subseteq BU$ and $BU \not\subseteq TD$ (TD and BU are incomparable),*
2. *Neither TD nor BU is closed under composition.*
3. *$BU \subset TD^2$ and $TD \subset BU^2$, where TD^2 and BU^2 denote the classes of compositions of two top-down and bottom-up tree transductions respectively.*

In fact, it is known [Eng82] that the above holds for any number of compositions of tree transductions. That is, BU^n is incomparable with TU^n, and vice versa, while $BU^n \subset TD^{n+1}$ and $TD^n \subset BU^{n+1}$, for any $n \in \mathbb{N}_+$.

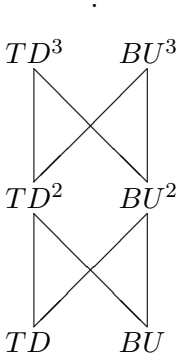


Figure 4: Inclusion diagram for compositions of td and bu tree transformations.

Chapter IV

Macro Tree Transducers

by Lovisa Pettersson

In this chapter we will define macro tree transducers, originally introduced and studied in [Eng80, CFZ82, EV85], and explain how they work. To this end, we will first mention some properties of MTTs, then formally define term rewrite systems and macro tree transducers. Finally, we will try to give an intuitive explanation with an example. For more information about macro tree transducers, the book [FV98] by Fülöp and Vogler may be of interest.

1 Introduction to Macro Tree Transducers

A macro tree transducer is a transducer more powerful than bottom-up and top-down tree transducers. They are related to other devices known from the literature or discussed in this report.

- They generalise top-down tree transducers. They are an extension of top-down tree transducers in the sense that every top-down tree transducer is a macro tree transducer, but not the other way around.
- They are related to context-free tree grammars. A context-free grammar is a generalisation of regular tree grammars, which allows non-terminals of rank ≥ 0 [Rou69].
- They are related to attribute grammars, which play an important part in computer language semantics. Attribute grammars were first introduced by Knuth [Knu68].

Macro tree transducers have been obtained by combining the ideas behind these devices.

2 Excursion: Term Rewrite Systems

Tree transducers and tree automata are special cases of term rewrite systems.[3] Term rewrite systems are used as a formal basis for functional and logic programming. Computationally, they are equivalent with Turing machines.

In the following, consider a ranked alphabet Σ and an alphabet V of symbols of rank 0, which we shall use as variables. A *substitution* is a mapping $\sigma\colon V \to$

[3] In these areas, *tree* and *term* are synonyms. Thus, in the terminology of these lecture notes, term rewrite systems rewrite trees.

$T_\Sigma(V)$. Such a substitution is often denoted by $[v_1/\sigma(v_1), \ldots, v_n/\sigma(v_n)]$ if $V = \{v_1 \ldots v_n\}$. Applying σ to a tree $t \in T_\Sigma(V)$ results in

$$t\sigma = \begin{cases} \sigma(t) & \text{if } t \in V \\ f[t_1\sigma, \ldots, t_k\sigma] & \text{if } t = f[t_1, \ldots, t_k], f \notin V. \end{cases}$$

Substitution is a small generalisation of the notation used in the previous chapter. If $V = X_k = \{x_1, \ldots x_k\}$ then $t\sigma = t[\![t_1, \ldots t_k]\!]$ where $\sigma = [x_1/t_1, \ldots, x_k/t_k]$. If $V = \{v\}$ and v occurs exactly once in t, then $t\sigma$ is often denoted $t \cdot t'$ where $t' = \sigma(v)$. This is a convenient notation, because we can write $s = t \cdot t'$ to "cut" an occurrence of a subtree t' out of a larger tree s, the remainder being t.

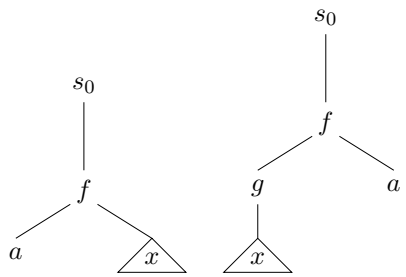
Example 1 (Substitution of variables) Let $t = f[x, g[a, y, y]]$ and consider the alphabet of variables $V = \{x, y\}$ and the substitution $[x/h[y], y/f[a, x]]$. The substitution may be read "x and y are substituted by $h[y]$ and $f[a, x]$, resp." Applying the substitution to t gives the output tree $t[x/h[y], y/f[a, x]] = f[h(y), g[a, f[a, x], f[a, x]]$. As was already pointed out in Chapter III, the operation is not performed recursively, so the substitution will not be applied to inserted subtrees.

Definition 2 (Term rewrite system) *A* term rewrite system *is a finite set R of rules of the form $l \to r$ such that:*

1. *$l, r \in T_\Sigma(V)$ and*
2. *all variables in r occur in l as well.*

We write $s \underset{R}{\to} t$ (for $s, t \in T_\Sigma$) if s and t can be decomposed as $s = s_0 \cdot l\sigma$ and $t = s_0 \cdot r\sigma$ for some substitution σ, a suitable tree s_0, and a rule $l \to r \in R$; $s_0 \cdot l\sigma$ is called the decomposition the step $s \underset{R}{\to} t$ is based on.

Example 3 Consider the tree $s_0[f[a, x]]$, which can be seen in Figure 5(a). The variable x represents a subtree $\sigma(x)$ in a particular context, such that the rule $(l \to r) = (f[a, x] \to f[g[x], a])$ applies. The resulting output tree will then be $s_0[f[g[x], a]]$, illustrated in Figure 5(b).



(a) The input tree (b) The output tree

Figure 5: The input- and output trees

When using term rewriting as a model of computation, rules from the given term rewrite system R are applied as long as possible. When there are no more rules in R that match the current tree, the computation terminates. In general, this process is non-deterministic. It may be interesting to note that it is undecidable whether a term rewrite system terminates or not, even for term rewrite systems containing only one rule [Dau92].

A term rewrite system may have the property of confluence. Confluence means that regardless of which rules are applied, the end result will always be the same. This means that the term rewrite system computes a partial function, yielding at most one result for every input, even though the computation as such may be non-deterministic.

A rule is *left linear* if each variable occurs just once in the left side. Similarly, it is *right linear* if each variable occurs at most once in the right side, and *linear* if each variable occurs just once in each side, that is, it is both left linear and right linear.

When a system is left-linear, there are no constraints on the input-subtrees. There can, for example, be no rules that forces two input-subtrees to be equal. When it is right-linear, there will be no duplications of the subtrees in the resulting trees.

3 Macro Tree Transducers – the Formal Definition

Macro tree transducers generalise top-down tree transducers by allowing states with rank ≥ 1. That allows the macro tree transducer to carry the result of finitely many sub-computations in each state and insert them into the tree at a later stage, allowing for far more powerful computations. In particular, this gives the possibility to have nested right-hand sides. With nested right-hand sides comes the requirement to decide in which order nodes are to be evaluated, because this may influence the result of the computation.

Definition 4 (Macro tree transducer) *A* macro tree transducer*, MTT for short, is a tuple* $mt = (\Sigma, \Sigma', Q, R, q_0)$ *where*

1. Σ *and* Σ' *are ranked alphabets of input- and output symbols,*
2. Q *is a ranked alphabet of states of rank* ≥ 1*,*
3. R *is a term rewrite system, where every rule* $l \to r$ *in* R *satisfies* $l = q[f[x_1, \ldots, x_k], y_1, \ldots, y_p]$ *for some* $q^{(p+1)} \in Q, f^{(k)} \in \Sigma$ *and* $r \in RHS_k^p$ *(see below), and*
4. $q_0^{(1)} \in Q$*.*

The set RHS_k^p *is defined inductively as follows:*

1. $T_{\Sigma'}(RHS_k^p) \subseteq RHS_k^p$*,*
2. $q'[x_i, t_1, \ldots, t_m] \in RHS_k^p$ *for* $q' \in Q^{(m+1)}, 1 \leq i \leq k$ *and* $t_1, \ldots t_k \in RHS_k^p$*, and*
3. $y_1, \ldots, y_p \in RHS_k^p$*.*

Definition 5 (Semantics of MTTs) *Let $mt = (\Sigma, \Sigma', Q, R, q_0)$ be an MTT. Given a tree $s \in T_\Sigma$, the computation of mt with input s starts with $q_0[s]$. There are three different computation modes:*

1. *An* unrestricted *computation step $t \Rightarrow_{mt} t'$ is a term rewrite step $t \underset{R}{\rightarrow} t'$.*
2. *An* IO*, Inside-Out, computation step $t \Rightarrow_{mt,IO} t'$ is a term rewrite step $t \underset{R}{\rightarrow} t'$ based on a decomposition $s \cdot s'$ of t such that s' contains only one state (namely, the root symbol).*
3. *An* OI*, Outside-In, computation step $t \Rightarrow_{mt,OI} t'$ is a term rewrite step $t \underset{R}{\rightarrow} t'$ based on a decomposition $s \cdot s'$ of t such that s does not contain any non-terminal on the path from the root to its unique variable.*

We let:

$$\begin{aligned} mt(s) &= \{s' \in T_{\Sigma'} \mid q_0[s] \underset{mt}{\overset{*}{\Rightarrow}} s'\}^4 \\ mt_{IO}(s) &= \{s' \in T_{\Sigma'} \mid q_0[s] \underset{mt,IO}{\overset{*}{\Rightarrow}} s'\} \\ mt_{OI}(s) &= \{s' \in T_{\Sigma'} \mid q_0[s] \underset{mt,OI}{\overset{*}{\Rightarrow}} s'\}. \end{aligned}$$

Note that, as a direct consequence of these definitions, $mt_{IO}(s) \subseteq mt_{unr}(s)$ and $mt_{OI}(s) \subseteq mt_{unr}(s)$. In fact, it can be shown that $mt_{unr}(s) \subseteq mt_{OI}(s)$. Thus we have $mt_{IO}(s) \subseteq mt_{OI}(s) = mt_{unr}(s)$.

3.1 Simulating MTTs with the YIELD Algebra

Macro tree transducers can be simulated using td transducers and the YIELD algebra. (See Definition 11 for the definition of the YIELD algebra.) In particular, this makes use of the substitution operation $subst_k$, where k is the number of variables y_i used in the rules of the MTT. When we want to simulate a state of rank $k+1$, the td transducer can output a tree of the form $subst_k[t_0, t_1 \ldots, t_k]$. When this tree is evaluated using a YIELD algebra $\mathcal{Y}$, it results in $Val_\mathcal{Y}(t_0)[\![Val_\mathcal{Y}(t_1), \ldots, Val_\mathcal{Y}(t_k)]\!]$, thus making it possible to simulate the ability of the MTT to insert $Val_\mathcal{Y}(t_1), \ldots, Val_\mathcal{Y}(t_k)$ into a computed subtree $Val_\mathcal{Y}(t_0)$. The basis for this recursion is $Val_\mathcal{Y}(f^{<l>}) = f[x_1, ..., x_l]$; see the following example.

Example 6 (MTT implementation of $Val_\mathcal{Y}$) Given a state q, and assuming that the computation has reached the leaves of the tree, applying the state q on $f^{<l>}$, with the parameters $y_1, \ldots, y_k$ would result in $q[f^{<l>}, y_1, \ldots, y_k] \rightarrow f[y_1, \ldots, y_l]$, where $l \leq k$. Similarly, applying q on $x_i^{<0>}$, would result in $q[x_i^{<0>}, y_1, \ldots, y_k] \rightarrow y_i$.
Note that this example is specialised, and MTTs are not forced to use these rules to implement a YIELD mapping.

Conversely, deterministic total MTTs can quite easily be simulated by td transducers and YIELD algebras. (For the other cases, see [FV98].) More precisely,

[4] Similarly to the notations used in previous chapters, $\overset{*}{\Rightarrow}$ denotes the transitive and reflexive closure of $\Rightarrow$.

we have $MTT_{dt} = Y \circ TD_{dt}$ where MTT_{dt} denotes all the deterministic total MTTs, Y denotes the set of all $Val_{\mathcal{Y}}$, where $\mathcal{Y}$ is a YIELD algebra, and TD_{dt} denotes the set of all deterministic total td transductions. Deterministic total is defined as that for each left side, there is exactly one rule in R.

Let us finally mention that, when MTTs are composed an arbitrary number of times, the difference between Inside out and Outside in MTTs vanishes. This can be formally written as $MTT^*_{IO} = (TD \cup Y)^* = MTT^*_{OI} = MTT^*$.

4 Example

In this example, the input tree is an arbitrary tree over $f^{(2)}$, $g^{(1)}$ and $a^{(0)}$. Figure 6 shows such an input tree, and Figure 7 shows the tree we want to transform it into. Compared with the input tree, all gs have been extracted and moved to the top of the output tree.

This transformation cannot be computed by any combination of top-down tree transducers, since it is unknown how many gs there are in the input tree when the first f is encountered. If one subtree is followed, the others are thrown away, and it will be unknown how many gs there are on those subtrees.

The transformation cannot be computed by bottom-up tree transducers either, because all the gs must be deleted on the way up, and remembering how many gs were deleted would require an infinite number of states.

So, neither top-down nor bottom-up nor any composition of finitely many of these tree transducers work. To produce the output tree, part of the original tree has to be rearranged. Top-down tree transducers can go down different branches, and replace or copy. But independent subtrees would remain independent. That is what can be overcome using macro tree transducers.

4.1 The Idea

We use states of rank ≥ 1, where the first subtree is the input (sub)tree to be processed and the other subtrees are temporarily stored output trees. The

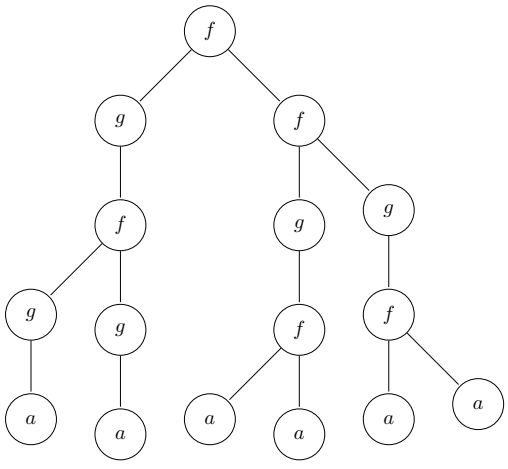


Figure 6: An input-tree

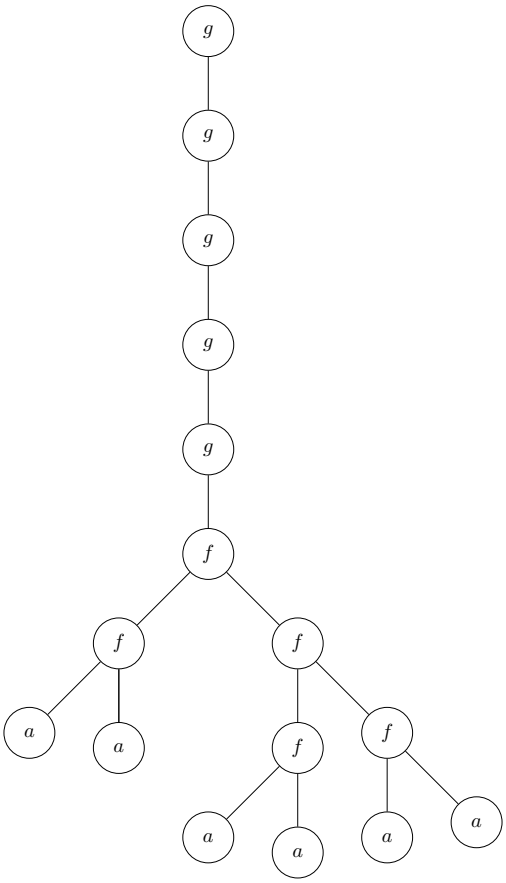


Figure 7: The output-tree

output trees do not need to be finished yet and can contain states that operate on other parts of the input tree.

Suitable rules for this example would be:

$$
\begin{array}{ll}
q_0[f[x_1, x_2]] & \to q_g[x_1, q_g[x_2, f[q[x_1], q[x_2]]]] \\
q_0[g[x_1]] & \to g[q_0[x_1]] \\
q_0[a] & \to a \\
q_g[f[x_1, x_2], y_1] & \to q_g[x_1, q_g[x_2, y_1]] \\
q_g[g[x_1], y_1] & \to g[q_g[x_1, y_1]]^5 \\
q_g[a, y_1] & \to y_1 \\
q[f[x_1, x_2]] & \to f[q[x_1], q[x_2]] \\
q[g[x_1]] & \to q[x_1] \\
q[a] & \to a
\end{array}
$$

The state q_0 is the start state, q_g is a state whose task it is to extract the gs and produce a monadic tree of them, and q is a state to remove the gs. The

[5] $q_g[g[x_1], y_1] \to q_g[x_1, g[y_1]]$ is also possible, but will, intuitively, reverse the order of the extracted gs.

ys represent parts of the output trees and cannot be inspected by macro tree transducers.

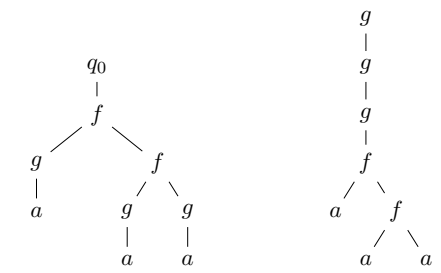
4.2 A Non-Deterministic Case

Assume we have the same macro tree transducer as before, but with the additional rule $q_g[g[x_1], y_1] \to gg[q_g[x_1, y_1], q_g[x_1, y_1]]$, where $gg^{(2)}$ is a symbol of rank 2. We will now have to decide if we want to keep the g, or turn it into a gg by copying the subtree. The MTT is now non-deterministic. Depending on which evaluation strategy is used, the non-determinism have different consequences.

1. Outside-In (OI). Here the copying takes place before the subtrees are evaluated, which means that the two copies of the same subtree do not have to be equal.
2. Inside-Out (IO). Here the subtrees are evaluated before the copying, assuring that the two copies of the same subtree are equal.
3. Unrestricted. Non-deterministically apply rules. Different states could have different evaluation strategies.

Example 7 (OI evaluation) Given the tree in Figure 8(a), doing an Outside-In evaluation without the additional rule would yield the tree seen in Figure 8(b), which is the expected result. But with the additional rule, subtrees may be copied, and the result is non-deterministic. Figure 9(a) shows a tree where the copying rule has been used. As can be seen, the copying is done before the subtrees are evaluated, and therefore the two subtrees do not have to be evaluated equally.

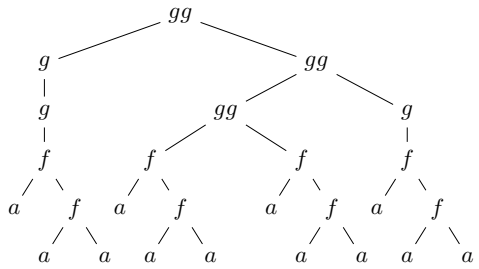
Example 8 (IO evaluation) Using the same input tree and rules as previously (without the additional rule) will yield the same result as the deterministic example. With the additional rule, the output tree may be as in Figure 9(b). Note that the subtrees are equal since they were evaluated before being copied.



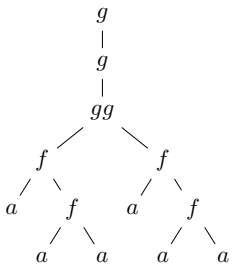
(a) The input tree to be evaluated

(b) The output tree (deterministic)

Figure 8: The input tree and the deterministic output tree



(a) The output tree (OI, non-deterministic)



(b) The output tree (IO, non-deterministic)

Figure 9: The results of OI and IO-evaluation.

Chapter V

Tree Automata on Unranked Trees

by Peter Winnberg

Unranked trees are more general than the ranked case, in the sense that removing the rank from the nodes in a ranked tree turns it into an unranked tree.

A major motivation for looking at unranked trees is the eXtensible Markup Language (XML). An XML document can be viewed as a tree. Consequently, an XML document class corresponds to a tree language. Several methods (e.g., Document Type Definitions (DTDs), XML Schema, and Relax NG) exist to specify the structure of a class of XML documents by defining the corresponding tree language.

The first section will look at unranked tree languages in general and introduce the necessary notation. The second section will look at an important method for converting between the ranked and unranked case, thus yielding a way to carry over results and techniques from the theory of ranked tree languages to unranked ones.

1 Definitions and Notation

In an unranked tree, the number of children of a given node is not determined by its label. The sequence of children of a given node is called a hedge, a term was coined by Bruno Courcelle [Cou78]. In other words, an unranked tree is a root symbol with a hedge of subtrees, this hedge being a (possibly empty) sequence of unranked trees.

Definition 1 (unranked trees and hedges) *Let Σ be an unranked alphabet. The sets of all* unranked trees *and of all* hedges *over Σ are the smallest sets T_Σ and H_Σ, respectively, such that*

- *every sequence of trees in T_Σ is in H_Σ (including the empty hedge ϵ), and*
- *for every $f \in \Sigma$ and $h \in H_\Sigma$, $f[h] \in T_\Sigma$.*

In the following, we may drop the attribute *unranked* when there is no risk of causing confusion. The simplest trees in T_Σ are single leaves, i.e., trees

consisting of a symbol $f \in \Sigma$ with the empty hedge below. Adopting the notation introduced after Definition 3 for the unranked case, this is the set $\Sigma(\emptyset)$.

Definition 2 (unranked tree automaton) *An* unranked tree automaton *is a tuple $A = (\Sigma, Q, R, F)$ consisting of*

- *an alphabet Σ,*
- *a finite set Q of states,*
- *a finite set R of rules of the form $f[L] \to q$ with $f \in \Sigma$, $L \subseteq Q^*$ a regular language and $q \in Q$, and*
- *$F \subseteq Q$ is the set of final states.*

For a tree $t = f[t_1, \ldots, t_n]$ and a state $q \in Q$, we let $t \to_A^* q$ if there are states $q_1, \ldots, q_n \in Q$ such that

- $t_i \to^* q_i$ for all $i, 1 \leq i \leq n$, and
- R contains a rule $f[L] \to q$ with $q_1, \ldots, q_n \in L$.

The language accepted by A is $L(A) = \{t \in T_\Sigma \mid t \to^* q \text{ for some } q \in F\}$.

Note that Definition 2 does not say anything about how R should be represented. One possible solution would be to represent each of the (finitely many) string languages L by a finite automaton that recognises L. However, it would be more concise to represent the complete set R using a *single* finite automaton. More precisely, we may demand that this automaton does the following for us. For each rule $f[L] \to q$, if the automaton is started in a special state $init_f$ associated with f, it ends up in state q upon reading $q_1 \cdots q_n \in L$. In fact, since there may be other rules $f[L'] \to q'$ also satisfying $q_1 \cdots q_n \in L'$, this automaton may non-deterministically end up in q' as well.

Definition 3 (stepwise tree automaton) *A* stepwise tree automaton *is an unranked tree automaton $A = (\Sigma, Q, R, F)$ in which the rules are specified as follows: A' is a (possibly nondeterministic) finite string automaton of the form $A' = (Q, Q, R', inits, \emptyset)$ where*

- *Q is the set of states, as well as the alphabet of the automaton,*
- *R' is the set of transitions of the automaton, and*
- *inits is a mapping $\Sigma \to Q$, which for each symbol $f \in \Sigma$ defines the initial state $init_f \in Q$ of the string automaton*

The set R of rules of the tree automaton A is defined by A' as follows:

$$R = \{f[L_{init_f,q}] \to q \mid f \in \Sigma,\, q \in Q\},$$

where $L_{init_f,q}$ denotes the string language accepted by A' if $init_f$ is chosen as the initial state and q is chosen as the final state.

Example 4 (XML representing people and groups) Consider an XML format that represents people and groups. The `G` element represents a group

and groups can be part of other groups. The `ID` element represents an identifier. Each group must have one identifier that can appear at any place within a `G` element.

The `P` element represents a person. The `F` element represents a given (first) name, of which each person has one or more. The `L` element represents a family (last) name. Each person has exactly one element of this type. Further, the `L` element must be the last child of the `P` element.
A document structured according to these rules may look like this:

```
<?xml version="1.0"?>
<G>
    <G>
        <P>
            <F/>
            <L/>
        </P>
        <ID/>
    </G>
    <P>
        <F/>
        <F/>
        <L/>
    </P>
    <P>
        <F/>
        <L/>
    </P>
    <ID/>
</G>
```

The corresponding tree is shown in Figure 10.

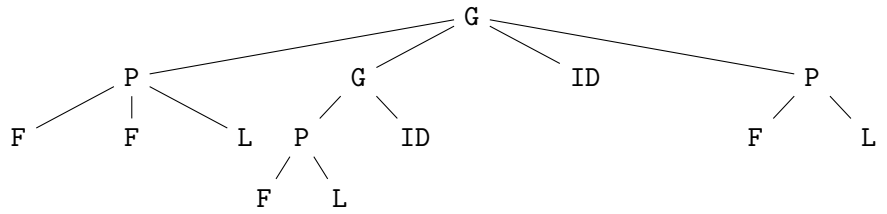


Figure 10: The XML document represented using a tree.

The stepwise tree automaton in Figure 11 accepts the tree language in the example. In the figure, for $f \in \{\mathtt{L}, \mathtt{L}, \mathtt{P}, \mathtt{ID}, \mathtt{G}\}$, the state $init_f$ is indicated by an f-labelled arrow that points to $init_f$. Thus, in this example, $init_f = q_0^f$ for all $f \in \{\mathtt{L}, \mathtt{L}, \mathtt{P}, \mathtt{ID}, \mathtt{G}\}$.

When the tree in Figure 10 is run trough it, the leaves will first be turned into the corresponding initial states. For example, when the left-most subtree of the root is processed, its hedge of leaves will be turned into $q_0^{\mathtt{F}} q_0^{\mathtt{F}} q_0^{\mathtt{L}}$. This is because processing a node labelled with f always results in the state q obtained by starting A' in state $init_f$ and running it on the string w of states assigned to the children of f. If f is a leaf, then w is the empty string. Consequently,

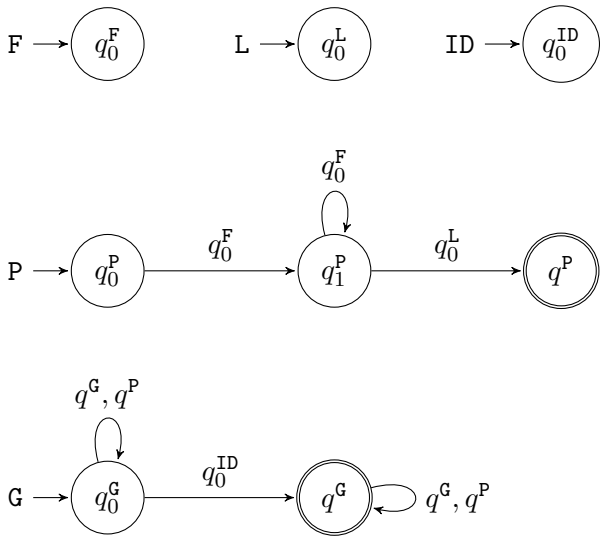


Figure 11: Stepwise tree automaton accepting the tree language corresponding to the XML document class discussed in Example 4.

the computation ends immediately in $q = init_f$. Going one level higher up, the string $q_0^{\mathtt{F}} q_0^{\mathtt{F}} q_0^{\mathtt{L}}$ will be processed, starting in the state $q_0^{\mathtt{P}}$, and the resulting state $q^{\mathtt{P}}$ will be assigned to the P-labelled node.

2 Converting the Unranked Case to the Ranked Case

The tree presented in Example 4 can be converted into a ranked tree by using a technique called currying to transform it. This is interesting because it allows us to use theory developed for ranked trees on converted unranked trees. The technique is named after the famous American logician Haskell Curry. The technique is known from mathematical logic, where it takes a function with multiple arguments and transforms it into function that can return functions. The actual function for doing the tree transformation is defined as follows, for $t = f[t_1, \ldots, t_n]$:

$$curry(t) = \begin{cases} f & \text{if } n = 0 \\ @[curry(f[t_1, \ldots, t_{n-1}]), curry(t_n)] & \text{if } n > 0. \end{cases}$$

Figure 12 shows the tree in Figure 10 after it has been transformed.

For every stepwise tree automaton (STA) A, the ranked bottom-up tree automaton A^r on curried trees is obtained by turning every transition $q_1 \rightarrow^{q_2} q$ in R' into $(q_1, q_2) \xrightarrow{@} q$ and adding all transitions $\xrightarrow{a} init_a$ for $a \in \Sigma$. This conversion of unranked tree automata to ranked ones leads gives rise to an interesting and useful observation.

Observation 5

- $L(A^r) = curry(L(A))$, *that is,* A^r *accepts exactly the curried versions of trees in* $L(A)$.

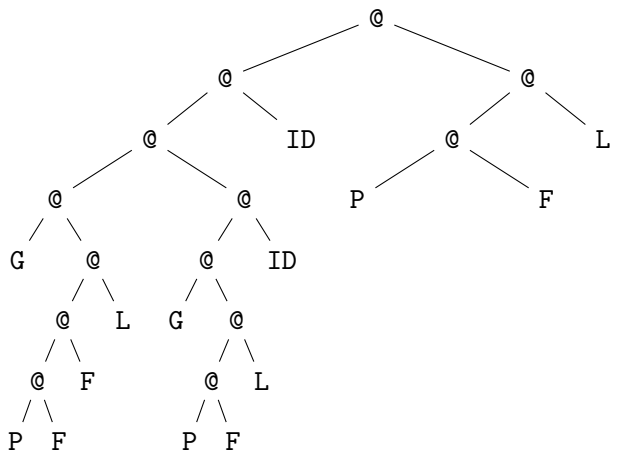


Figure 12: The tree on page 31 converted to a binary tree.

- *The mapping $A \mapsto A^r$ is bijective.*
- *A^r is deterministic if and only if A is deterministic.*

Since the correspondence established by the conversion of unranked into ranked tree automata is bijective, there is exactly one STA A that corresponds to the ranked tree automaton A^r and vice versa. As a consequence STA can be determinised, and deterministic STA can be minimised in polynomial time. Minimised STA are unique, so it is possible to see if two STA recognise the same language by determinising, minimising, and comparing them.

References

[Bak79] Brenda S. Baker. Composition of top-down and bottom-up tree transductions. *Information and Control*, 41:186–213, 1979.

[Bra68] Walter S. Brainerd. The minimalization of tree automata. *Information and Computation*, 13:484–491, 1968.

[Bra69] Walter S. Brainerd. Tree generating regular systems. *Information and Control*, 14:217–231, 1969.

[CDG+07] Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Christof Löding, Denis Lugiez, Sophie Tison, and Marc Tommasi. *Tree Automata Techniques and Applications*. Internet publication available at `http://tata.gforge.inria.fr`, 2007. Release October 2007.

[CFZ82] Bruno Courcelle and Paul Franchi-Zannettacci. Attribute grammars and recursive program schemes I, II. *Theoretical Computer Science*, 17:163–191, 235–257, 1982.

[Cou78] Bruno Courcelle. A representation of trees by languages I. *Theoretical Computer Science*, 6:255–279, 1978.

[Dau92] Max Dauchet. Simulation of turing machines by a regular rewrite rule. *Theoretical Computer Science*, 103(2):409–420, 1992.

[Don65] John E. Doner. Decidability of the weak second-order theory of two successors. *Notices of the American Mathematical Society*, 12:365–368, 1965.

[Don70] John E. Doner. Tree acceptors and some of their applications. *Journal of Computer and System Sciences*, 4:406–451, 1970.

[Dre06] Frank Drewes. *Grammatical Picture Generation – A Tree-Based Approach*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2006.

[Eng75] Joost Engelfriet. Bottom-up and top-down tree transformations – a comparison. *Mathematical Systems Theory*, 9:198–231, 1975.

[Eng80] Joost Engelfriet. Some open questions and recent results on tree transducers and tree languages. In R. V. Book, editor, *Formal Language Theory: Perspectives and Open Problems*, pages 241–286. Academic Press, 1980.

[Eng82] Joost Engelfriet. Three hierarchies of transducers. *Mathematical Systems Theory*, 15:95–125, 1982.

[EV85] Joost Engelfriet and Heiko Vogler. Macro tree transducers. *Journal of Computer and System Sciences*, 31:71–146, 1985.

[FV98] Zoltán Fülöp and Heiko Vogler. *Syntax-Directed Semantics: Formal Models Based on Tree Transducers*. Springer, 1998.

[FV09] Zoltán Fülöp and Heiko Vogler. Weighted tree automata and tree transducers. In Werner Kuich, Manfred Droste, and Heiko Vogler, editors, *Handbook of Weighted Automata*, chapter 9, pages 313–403. Springer, 2009.

[GS84] Ferenc Gécseg and Magnus Steinby. *Tree Automata*. Akadémiai Kiadó, 1984.

[GS97] Ferenc Gécseg and Magnus Steinby. Tree languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages.* Vol. 3: *Beyond Words*, chapter 1, pages 1–68. Springer, 1997.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2:127–145, 1968.

[MW67] Jorge Mezei and Jesse B. Wright. Algebraic automata and context-free sets. *Information and Control*, 11:3–29, 1967.

[NP92] Maurice Nivat and Andreas Podelski, editors. *Tree Automata and Languages*. Elsevier, 1992.

[Rou68] William C. Rounds. *Trees, Transducers and Transformations*. PhD thesis, Stanford University, 1968.

[Rou69] William C. Rounds. Context-free grammars on trees. In *Proceedings of the 1st Annual ACM Symposium on Theory of Computing (STOC)*, pages 143–148, 1969.

[Rou70] William C. Rounds. Mappings and grammars on trees. *Mathematical Systems Theory*, 4:257–287, 1970.

[Tha67] James W. Thatcher. Characterizing derivation trees of context-free grammars through a generalization of finite automata theory. *Journal of Computer and System Sciences*, 1:317–322, 1967.

[Tha70] James W. Thatcher. Generalized^2 sequential machine maps. *Journal of Computer and System Sciences*, 4:339–367, 1970.

[Tha73] James W. Thatcher. Tree automata: an informal survey. In A.V. Aho, editor, *Currents in the Theory of Computing*, pages 143–172. Prentice Hall, 1973.

[TW68] James W. Thatcher and Jesse B. Wright. Generalized finite automata theory with an application to a decision-problem of second-order logic. *Mathematical Systems Theory*, 2:57–81, 1968.