Taxonomies of Regular Tree Algorithms

Loek Cleophas[1] and Kees Hemerik[2]

[1] FASTAR/Espresso Research Group, Department of Computer Science,
University of Pretoria, 0002 Pretoria, Republic of South Africa,
http://www.fastar.org
[2] Software Engineering & Technology Group, Department of Mathematics and Computer Science,
Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands,
http://www.win.tue.nl/set
loek@loekcleophas.com, c.hemerik@tue.nl

Abstract. Algorithms for acceptance, pattern matching and parsing of regular trees and the tree automata used in these algorithms have many applications, including instruction selection in compilers, implementation of term rewriting systems, and model checking. Many such tree algorithms and constructions for such tree automata appear in the literature, but some deficiencies existed, including: inaccessibility of theory and algorithms; difficulty of comparing algorithms due to variations in presentation style and level of formality; and lack of reference to the theory in many publications. An algorithm taxonomy is an effective means of bringing order to such a field. We report on two taxonomies of regular tree algorithms that we have constructed to deal with the deficiencies. The complete work has been presented in the PhD thesis of the first author.

Keywords: tree acceptance, tree pattern matching, tree automata, algorithm taxonomies

1 Introduction

We consider the field of *regular tree languages* for *ordered, ranked trees*.[1] This field has a rich theory, with many generalizations from the field of regular string languages, and many relations between the two [9,10,12,14]. Parts of the theory have broad applicability in areas as diverse as instruction selection in compilers, implementation of term rewriting systems, and model checking.

We focus on algorithmic solutions to three related problems in the field, i.e. *tree acceptance, tree pattern matching* and *tree parsing*. Many such algorithms appear in the literature, but unfortunately some deficiencies exist, including:

1. Inaccessibility of the theory and algorithms, as they are scattered over the literature and few or no (algorithm oriented) overview publications exist.
2. Difficulty of comparing the algorithms due to differences in presentation style and level of formality.
3. Lack of reference to the theory and of correctness arguments in publications of practical algorithms.

A *taxonomy*—in a technical sense made more precise below—is an effective means of bringing order to such a subject. A taxonomy is a systematic classification of problems and solutions in a particular (algorithmic) problem domain. We have constructed two such taxonomies, one for tree acceptance algorithms and one for tree pattern matching ones.

[1] An example of such a language as defined by a regular tree grammar can be found in Section 4.

Loek Cleophas, Kees Hemerik: Taxonomies of Regular Tree Algorithms, pp. 146–159.
Proceedings of PSC 2009, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04403-2

A few more practical deficiencies existed as well: no large and coherent collection of implementations of the algorithms existed; and for practical applications it was difficult to choose between algorithms. We therefore designed, implemented, and benchmarked a highly coherent *toolkit* of most of these algorithms as well. Taxonomies also form a good starting point for the construction of such algorithmic toolkits.

In the past, taxonomies and/or toolkits of this kind have been constructed for e.g. sorting [3,11], garbage collection [17], string pattern matching, finite automata construction and minimization [21,22].

In this paper we focus on one of our taxonomies, and comment only briefly on the other one and on the toolkit. The complete work has been presented in the PhD thesis of the first author [9]. For more details we refer to this thesis and to recent shorter publications [5,6,18].

Section 2 gives a brief introduction to taxonomies as we consider them. In Section 3 we outline the structure of our taxonomy of algorithms for tree acceptance and briefly compare it to the one for tree pattern matching. Afterwards we focus on the one for tree acceptance. Definitions of tree and tree grammar related notions are given in Section 4. The main branches of the taxonomy for tree acceptance are discussed in Sections 5–8. Section 9 briefly discusses some other parts of the work, namely the toolkit and accompanying graphical user interface and the benchmarking experiments performed with them. We end the paper with some concluding remarks in Section 10.

2 Taxonomies

In our technical sense a taxonomy is a means of ordering a set of algorithmic problems and their solutions. Each node of the taxonomy graph is a pair consisting of (a specification of) a problem and an algorithm solving the problem. For each (problem, algorithm) pair the set of essential details is determined. In general, there are two kinds of details: *problem details*, which restrict the problem, and *algorithm details*, which restrict the algorithm (e.g. by making it more deterministic). The root of the taxonomy graph contains a high-level algorithm of which the correctness is easily shown. A branch in the graph corresponds to addition of a detail in a correctness preserving way. Hence, the correctness of each algorithm follows from the details on its root path and the correctness of the root.

Construction of an algorithm taxonomy is a bottom-up process. A literature survey of the problem domain is performed to gather algorithms. The algorithms are rephrased in a common presentation style and analyzed to determine their essential details. When two algorithms differ only in a few details, abstracting over those details yields a common ancestor. Repeating this abstraction process leads to the main structure of a taxonomy graph. Considering new combinations of details may lead to discovery of new algorithms. Eventually the taxonomy may be presented in a top-down manner.

Several taxonomies of this kind appear in the literature. Broy and Darlington each constructed one of sorting algorithms [3,11]. Jonkers [17] constructed a taxonomy of garbage collection algorithms and also developed a general theory about algorithm taxonomies. Watson [21] applied the method to construct taxonomies for string pattern algorithms and for the construction and minimization of finite automata. Both in subject and in style our work is closest to Watson's.

3 Overview of the Taxonomies of Regular Tree Algorithms

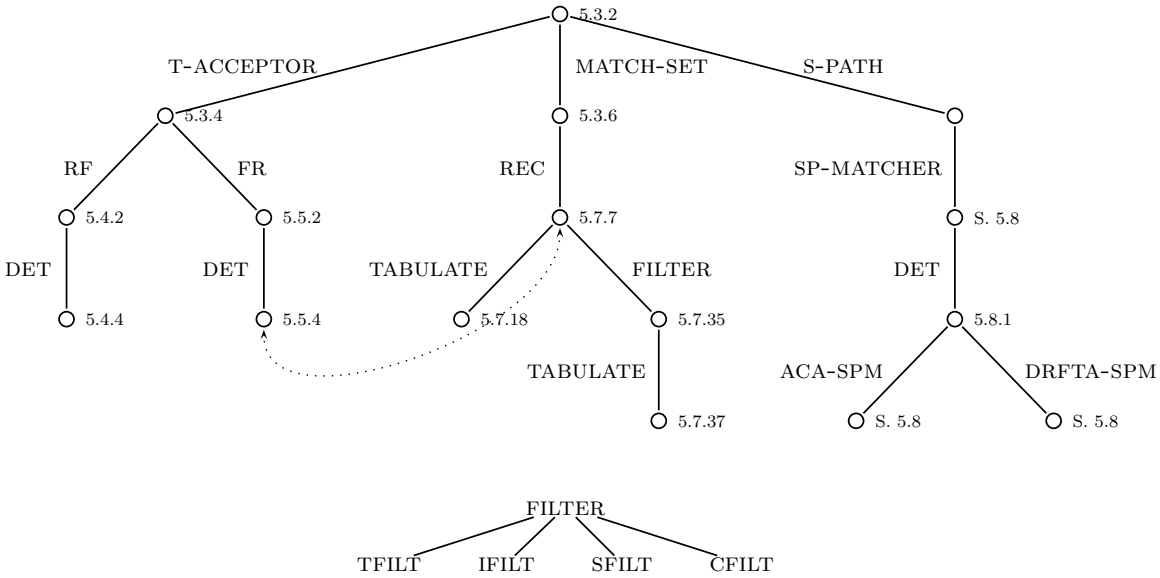

Figure 1. Tree acceptance taxonomy. Each node is labeled with its corresponding algorithm or section (S.) number in [9]. Constructions for tree acceptors used in algorithms of branch (T-ACCEPTOR) are not depicted. The bottom part of the figure shows the four possible filters that can be used for detail FILTER.

The tree acceptance (aka language membership, membership) problem as we consider it is the following: Given a regular tree grammar and a subject tree, determine whether the tree is an element of the language defined by the grammar. Figure 1 depicts the taxonomy of algorithms we have constructed for this problem. The edge labels correspond to details, explained in Table 1.

In the taxonomy graph, three main subgraphs can be distinguished. The first subgraph (detail T-ACCEPTOR and below) contains all algorithms based on the correspondence between regular tree grammars and finite tree automata. For every regular tree grammar an undirected finite tree automaton can be constructed, which accepts exactly the trees generated by the grammar. By adding more detail, viz. a direction (detail FR: frontier-to-root or detail RF: root-to-frontier) or determinacy (detail DET) more specific constructions are obtained. The acceptance algorithms from this part of the taxonomy are described in more detail in Section 5, while the tree automata constructions used in them are discussed in Section 6.

The second subgraph (detail MATCH-SET and below) contains all algorithms based on suitably chosen generalizations of the relation $S \stackrel{*}{\Rightarrow} t$ (where $\stackrel{*}{\Rightarrow}$ indicates derivation in zero or more steps (see Section 4), S is the start symbol of the grammar and t is the subject tree). For each subtree of t, they compute a set of *items* from which t may be derived, a so-called *match set*. Tree t is accepted if and only if its match set contains S. The algorithms in this subgraph of the taxonomy differ in the item set used and in how the match sets are computed. This part of the taxonomy is described in more detail in Section 7.

T-ACCEPTOR	Use a tree automaton accepting the language of a regular tree grammar to solve the language membership problem.
RF	Consider the transition relations of the tree automaton used in an algorithm to be directed in a root-to-frontier or top-down direction.
FR	Consider the transition relations of the tree automaton used in an algorithm to be directed in a frontier-to-root or bottom-up direction.
DET	Use a deterministic version of an automaton.
MATCH-SET	Use an item set and a match set function to solve the tree acceptance/language membership problem. Such an item set is derived from the productions of the regular tree grammar and the match set function indicates from which of these items a tree is derivable.
REC	Compute match set values recursively, i.e. compute the match set values for a tree from the match set values computed for its direct subtrees.
FILTER	Use a filtering function in the computation of match set function values. Before computing the match set for a tree, such a filtering function is applied to the match sets of its direct subtrees.
TABULATE	Use a tabulated version of the match set function (and of the filter functions, if filtering is used), in which a bijection is used to identify match sets by integers.
S-PATH	Uniquely decompose production right hand sides into stringpaths. Based on matching stringpaths, production right hand sides and nonterminals deriving the subject tree can be uniquely determined and tree acceptance can thus be solved.
SP-MATCHER	Use an automaton as a pattern matcher for a set of stringpaths in a root-to-frontier or top-down subject tree traversal.
ACA-SPM	Use an (optimal) Aho-Corasick automaton as a stringpath matcher and define transition and output functions in terms of that automaton.
DRFTA-SPM	Use a deterministic root-to-frontier tree automaton as a stringpath matcher and define transition and output functions in terms of that automaton.

Table 1.

The third subgraph (detail SP-MATCHER and below) contains algorithms based on the decomposition of items into so-called *stringpaths* and subsequent use of string matching techniques. Based on stringpath matches found, matches of items and hence essentially the match sets mentioned previously are computed for each subtree of t. Section 8 gives a brief explanation of this taxonomy part.

As our focus in this paper is on the tree acceptance taxonomy and the algorithms and constructions included in it, we do not formally define the tree pattern matching problem. Figure 2 shows the taxonomy of tree pattern matching algorithms. Although we do not explicitly give the meaning of the details used, it should be clear that the taxonomies for tree acceptance and tree pattern matching have much in common. Techniques such as the subset construction, match sets, and stringpaths are used in both. This is not surprising: the two problems are closely related, and some kinds of tree acceptors can be turned into tree pattern matchers (or vice versa) with little effort. The same phenomenon can be observed in acceptors and pattern matchers for string languages.

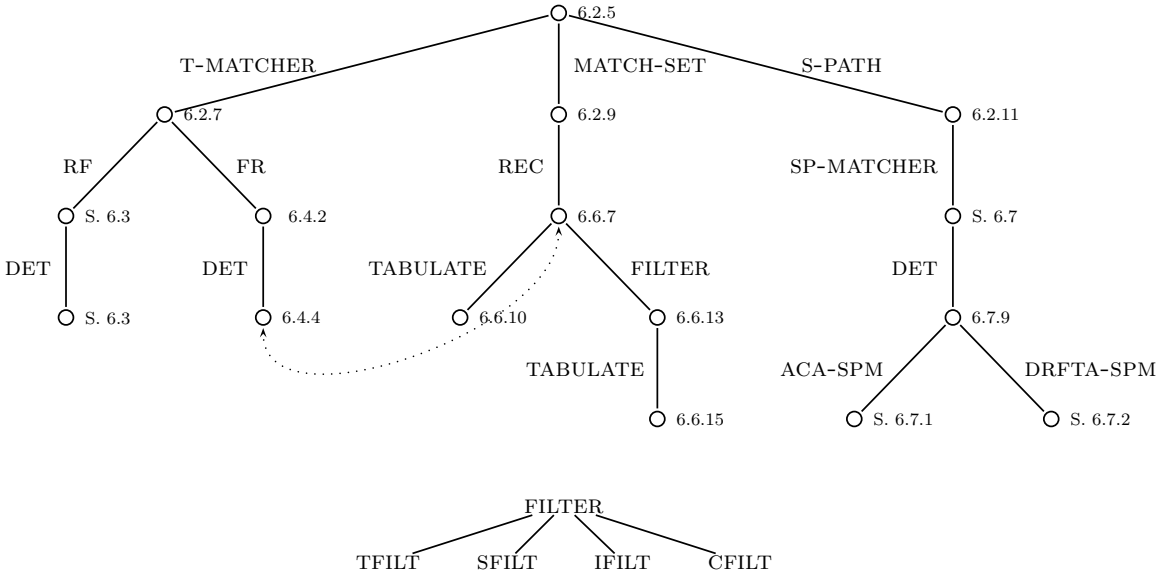

Figure 2. Tree pattern matching taxonomy. Each node is labeled with its corresponding algorithm or section (S.) number in [9]. Constructions for tree pattern matchers used in algorithms of branch (T-MATCHER) are not depicted. The bottom part of the figure shows the four possible filters that can be used for detail FILTER.

4 Notation and definitions

We use $\mathbb{B}$ and $\mathbb{N}$ to denote the booleans and the natural numbers. We use notation $\langle \textbf{Set}\ a : R(a) : E(a) \rangle$ for the set of expressions $E(a)$ for which a satisfies range predicate $R(a)$.

Many of the other notations and definitions we use are related to regular tree language theory and to a large extent generalizations of familiar ones from regular string language theory. To aid readers unfamiliar with this theory, we briefly introduce the concepts needed in the rest of this paper. Readers may want to consult e.g. [9,10,12,14] for more detail.

Let Σ be an *alphabet*, and $r \in \Sigma \mapsto \mathbb{N}$. Pair (Σ, r) is a *ranked alphabet*, r is a *ranking function*, and for all $a \in \Sigma$, $r(a)$ is called the *rank* or *arity* of a. (The ranking function indicates the number of child nodes a node labeled by a particular symbol will have.) We use Σ_n for $0 \leq n$ to indicate the subset of Σ of symbols with arity n.

Given a ranked alphabet (Σ, r), the set of *ordered, ranked trees* over this alphabet, set $Tr(\Sigma, r)$, is the smallest set satisyfing

1. $\Sigma_0 \subseteq Tr(\Sigma, r)$, and
2. $a(t_1, \ldots, t_n) \in Tr(\Sigma, r)$ for all $t_1, \ldots, t_n \in Tr(\Sigma, r), a \in \Sigma$ such that $r(a) = n \neq 0$.

As a running example, we assume (Σ, r) to be $\{(a, 2), (b, 1), (c, 0), (d, 0)\}$, i.e. consisting of symbols a, b, c and d with rank $2, 1, 0$ and 0. Trees in $Tr(\Sigma, r)$ include for example c, $a(b(c), d)$ and $a(a(b(c), c), d)$.

A *regular tree grammar* (RTG) G is a 5-tuple $(N, \Sigma, r, Prods, S)$ where N and Σ are disjoint alphabets (the *nonterminals* and *terminals*), $(N \cup \Sigma, r)$ is a ranked alphabet in which all nonterminals have rank 0, $Prods \subseteq N \times Tr(N \cup \Sigma, r)$ is the

finite set of *productions*, and $S \in N$ (the *start symbol*). We use LHS and RHS for left hand side and right hand side (of a production), and use $RHS(Prods)$ for the set of production RHSs.

Given a grammar G, we use $\Rightarrow$ for a derivation step, in which a nonterminal is replaced by a corresponding production RHS. The reflexive and transitive closure of $\Rightarrow$ is denoted by $\overset{*}{\Rightarrow}$. The subset of $Tr(\Sigma, r)$ derivable from S is denoted $\mathcal{L}(G)$. For technical reasons, we introduce the *augmented* grammar G' for a grammar G, defined by $G' = (N \cup \{S'\}, \Sigma, r \cup \{(S', 0)\}, Prods \cup \{S' \mapsto S\}, S')$ where S' is a fresh symbol.

In this paper, we assume an example grammar $G_1 = (N, \Sigma, r, Prods, S)$ with $N = \{S, B\}$, r and Σ as before, and with $Prods$ defined as $\{S \mapsto a(B, d),\ \ S \mapsto a(b(c), B),\ S \mapsto c,\ \ B \mapsto b(B),\ \ B \mapsto S,\ \ B \mapsto d\}$. We assume G to be the corresponding augmented grammar.

5 Algorithms based on Tree Automata

The first subgraph of the taxonomy deals with algorithms for tree acceptance that are based on correspondences between regular tree grammars and finite tree automata. The theoretical basis for this correspondence is well-known and generalizes a similar correspondence between regular string grammars and finite string automata. To ease understanding we briefly outline how the generalization works.

It is well known that the theory of regular tree languages generalizes that of regular string languages [9,10,12,14]. This is not surprising: any string $a_0 \cdots a_{n-1}$ can be seen as a special kind of regular tree, viz. one consisting of n unary nodes, each labeled with a symbol a_i of rank 1, closed by a nullary node marked with a symbol of rank 0. Notions from finite automata for strings can be generalized to the tree case as well, although this requires a particular view of such automata. Suppose that a particular string automaton goes through a state sequence $q_0, \ldots, q_n$ when presented the string $a_0 \cdots a_{n-1}$. This means that for each $i : 0 \leq i < n$ the pair of states (q_i, q_{i+1}) must be in the transition relation of symbol a_i. We can summarize the transition sequence by the following alternation of states and symbols: $q_0 a_0 \cdots a_{n-1} q_n$. In other words, the positions in the string have been consistently annotated with states $q_0, \ldots, q_n$. The language accepted by the automaton can be defined as the set of strings that can be consistently annotated in this way, such that q_0 and q_n are initial and final states.

This view can easily be generalized to ordered, ranked trees: each node is annotated with a state, and for each node labeled with a symbol a of rank n, the state q_0 assigned to that node and the states $q_1, \ldots, q_n$ of the n direct subnodes should be such that the tuple $(q_0, (q_1, \ldots, q_n))$ is in the transition relation of symbol a. Note that this simplifies to $(q_0, ())$ for symbols of rank 0. (Hence, taking a frontier-to-root or bottom-up view on tree automata, no equivalent for a string automaton's initial states is needed; no equivalent for a string automaton's final states is needed when taking a root-to-frontier or top-down view.) A tree is *accepted* by a finite tree automaton if and only if it can be consistently annotated such that the state assigned to the root is a so-called *root accepting* state. This motivates the following definition:

Definition 1. *A* (finite) tree automaton (TA) *M is a 5-tuple $(Q, \Sigma, r, R, Q_{ra})$ such that Q is a finite set, the* state set*; (Σ, r) is a ranked alphabet; $R = \{R_a | a \in \Sigma\} \cup R_\varepsilon$ is the set of transition relations (where $R_\varepsilon \subseteq Q \times Q$ and $R_a \subseteq Q \times Q^n$, for all $a \in \Sigma$ with $r(a) = n$); and $Q_{ra} \subseteq Q$ is the set of* root accepting states.

Many important theorems carry over from regular string grammars and automata to the tree case as well. In particular:

Theorem 2. *For every regular tree grammar G there exists a tree automaton M such that $\mathcal{L}(G) = \mathcal{L}(M)$.*

This theorem justifies the following algorithm as a solution for tree acceptance:

Algorithm 3 (T-ACCEPTOR)

$[\![$ **const** $G = (N', \Sigma, r', Prods', S')$: augmented RTG;
$t : Tr(\Sigma, r)$;
var $b : \mathbb{B}$
| **let** $M = (Q, \Sigma, r, R, Q_{ra})$ be a TA such that $\mathcal{L}(M) = \mathcal{L}(G)$;
$b := t \in \mathcal{L}(M)$
{ $b \equiv t \in \mathcal{L}(G)$ }
$]\!]$

This abstract and rather trivial algorithm forms the root of the part of the taxonomy graph containing all algorithms based on tree automata. Note that it does not specify *how* $t \in \mathcal{L}(M)$ is determined. It could consider all state assignments to t respecting the transition relations R, and determine whether an accepting one exists.

To obtain more specific and more practical algorithms, the automata and hence the state assignments can be considered as directed ones (detail FR: frontier-to-root aka bottom-up or detail RF: root-to-frontier aka top-down). This results in (the use of) an *ε-nondeterministic frontier-to-root* TA (εNFRTA) and *ε-nondeterministic root-to-frontier* TA (εNRFTA).

Restricting the directed automata to the case without ε-transitions, we obtain the ε-less TA and (ε-less) NRFTA and NFRTA. As with string automata, ε-transitions can be removed by a straightforward transformation. The use of the resulting automata slightly simplifies the acceptance algorithms.

5.1 FR: Frontier-to-Root Tree Acceptors

For (ε)NFRTAs, a recursive acceptance function $RSt \in Tr(\Sigma, r) \mapsto \mathcal{P}(Q)$ can be defined. This function yields the states assigned to a tree's root node based on those assigned to that node's child nodes. A subject tree t is then accepted if and only if at least one accepting state occurs in state set $RSt(t)$.

Restricting the directed R_a of the (ε-less) NFRTA to be single-valued functions, we obtain the deterministic DFRTA. A subset construction $\text{SUBSET}_{\text{FR}}$ can be given, similar to that for string automata, to obtain a DFRTA for an (ε)NFRTA. The use of a DFRTA leads to the straightforward Algorithm 4 given below.

Algorithm 4 (T-ACCEPTOR, FR, DET)

```
|[ const G = (N', Σ, r', Prods', S') : augmented RTG;
          t : Tr(Σ, r);
  var b : 𝔹
| let M = (Q, Σ, r, R, Q_ra) be a DFRTA such that 𝓛(M) = 𝓛(G);
  b : = Traverse(t) ∈ Q_ra
  { b ≡ t ∈ 𝓛(G) }

  func Traverse(st : Tr(Σ, r)) : Q =
  |[
  | let a = st(ε);
    { st = a(st_1, ..., st_n) where n = r(a) }
    Traverse : = R_a( Traverse(st_1), ..., Traverse(st_n))
  ]|{  Post: {Traverse} = RSt(st) }
]|
```

5.2 RF: Root-to-Frontier Tree Acceptors

For root-to-frontier automata, we can define a root-to-frontier acceptance function $Accept \in Tr(\Sigma, r) \times Q \mapsto \mathbb{B}$ indicating whether an accepting computation starting from some state exists for a tree. In the resulting Algorithm (T-ACCEPTOR, RF) (not given here), the value of this function is computed by possibly many root-to-frontier subject tree traversals (starting from each of the root accepting states).

As with FRTAs, RFTAs can be restricted to ε-less ones and further to deterministic ones. Since DRFTAs are known to be less powerful than other TA kinds, algorithms using DRFTAs cannot solve the acceptance problem for each input grammar. We refer the reader to [9] for more information on algorithms using RFTAs to directly solve the tree acceptance problem.

In Section 8 we briefly discuss how DRFTAs can be used for so-called stringpath matching. Since there is a one-to-one correspondence between a tree and its set of stringpaths, DRFTAs can thus be used to solve the tree acceptance problem, albeit indirectly.

6 Construction of tree automata

Nowhere in Section 5 did we specify *how* the tree automata M, which are used in Algorithm (T-ACCEPTOR) and derived algorithms, are to be constructed. Such constructions can be considered separately, as we do in this section.

Algorithm (T-ACCEPTOR) and derived ones use TAs M such that $\mathcal{L}(M) = \mathcal{L}(G)$. Depending on the algorithm, the acceptor may need to be undirected or directed RF or FR, and directed ones may need to be nondeterministic or deterministic. The constructions differ in a number of aspects:

- Which *item set* is used to construct states: one containing *all* subtrees of production RHSs, or one just containing all nonterminals as well as the *proper* subtrees among RHSs,
- whether ε-*transitions* are present or not—the latter indicated by label REM-ε ,

– whether automata are *undirected*, *root-to-frontier* (aka top-down) or *frontier-to-root* (aka bottom-up), and
– whether ε-less directed automata are *deterministic* or not.

By combining choices for these aspects, twenty four constructions for tree acceptors can be obtained. Roughly half are treated in [9, Chapter 6], seeming most interesting because they occur in the literature or because they lead to ones that do.

For each construction in our taxonomy, the discussion in [9, Chapter 6] defines the state set, root accepting state set and transition relation are defined; and usually gives an example and a discussion of correctness and of related constructions and literature. Presenting all of the constructions in such a similar, uniform and precise way facilitates understanding and comparing the different constructions.

To further simplify understanding and comparison, the constructions are identified by sequences of detail labels. For example, the first construction, Construction (TGA-TA:ALL-SUB), is a basic construction for undirected TAs. Its state set corresponds to all subtrees of production RHSs, while its transitions encode the relations between (tuples of) such states, based on the relation between a tree and its direct subtrees and the relation between a production LHS and RHS.

We cannot present the constructions here in detail, but restrict ourselves to describing them briefly and showing how constructions from the literature are included. We emphasize that our taxonomy presents all of them together and relates all of them for the first time.

– The basic Construction (TGA-TA:ALL-SUB) described above does not explicitly appear in the literature, but its FR and RF versions appear in van Dinther's 1987 work [20].
– Applying REM-ε results in Construction (TGA-TA:ALL-SUB:REM-ε) for automata isomorphic to those constructed by Ferdinand et al. (1994) [13]. This detail makes states corresponding to certain full RHSs unreachable and therefor useless.
– To prevent such states from occurring, a state set containing only nonterminals and proper subtrees of RHSs can be used instead. Of the resulting Construction (TGA-TA:PROPER-N:REM-ε),
  • an undirected version appears in Ferdinand, Seidl and Wilhelm's 1994 paper [13] and later in Wilhelm & Maurer [23]. Somewhat surprisingly, the construction in its general form apparently did not occur in the literature before 1994.

  It is well known however that every RTG can easily be transformed into one with productions of the form $A \mapsto a(A_1, \ldots, A_n)$ only (by introducing fresh nonterminals and productions). For such RTGs,
  • an FR directed version already appeared in Gecseg and Steinby's [14, Lemma 3.4] in 1984.

  It is also straightforward to transform any RTG into one with productions of the form given above and of the form $A \mapsto B$ (i.e. additionally allowing unit productions). For such RTGs,
  • an FR directed version of Construction (TGA-TA:PROPER-N:REM-ε) already appears in Brainerd's 1960s work [2] and again in [20], and
  • an RF directed version appears in Comon et al. 's online work [10].
– Constructions (TGA-TA:ALL-SUB:REM-ε:RF:SUBSET$_{\text{RF}}$) and (TGA-TA:PROPER-N:-REM-ε:RF:SUBSET$_{\text{RF}}$), which are derived constructions resulting in DRFTAs, do not appear in the literature, probably due to the restricted power of such automata.

For a specific subclass of RTGs for which DRFTAs can be constructed, a variant resulting in tree *parsers* based on such DRFTAs is presented in [20].
- A construction for DFRTAs which uses all RHSs for state set construction—i.e. Construction (TGA-TA:ALL-SUB:REM-ε:FR:SUBSET$_{\text{FR}}$)—appears in [15]. The encompassed subset construction constructs the reachable subsets only, with an explicit sink state for the empty set. The presentation mostly disregards the automata view and uses the recursive match set view of Section 7. It was inspired by and gives a more formal version of the initial construction presented in Chase's 1987 paper [4].
- A construction for DFRTAs which uses only nonterminals and proper subtrees of RHSs—Construction (TGA-TA:PROPER-N:REM-ε:FR:SUBSET$_{\text{FR}}$)—appears in [13, Section 6] and in [23, Sections 11.6–11.7].

7 Algorithms based on Match Sets

In this section we consider the second subgraph of the taxonomy. Algorithms in this part solve the tree acceptance problem, i.e. $S \stackrel{*}{\Rightarrow} t$, by suitably chosen generalizations of relation $\stackrel{*}{\Rightarrow}$. First, from the tree grammar a set of *Items* is constructed, e.g. the set of subtrees of right hand sides of productions of the grammar. Then, for the subject tree t, a so-called match set $MS(t)$ is computed, the set of all $p \in Items$ for which $p \stackrel{*}{\Rightarrow} t$ holds. Tree t is accepted if and only if $S \in MS(t)$.

Algorithms in this part of the taxonomy differ in the set *Items* used and in how function MS is computed. The first algorithm, Algorithm (MATCH-SET), does not specify *how* to compute function MS.

Function MS can effectively be computed recursively over a subject tree, i.e. by a scheme of the form $MS(a(t_1, \ldots, t_n)) = \mathcal{F}(MS(t_1), \ldots, MS(t_n))$. Function $\mathcal{F}$ composes and filters items for $MS(a(t_1, \ldots, t_n))$ from those in the match sets $MS(t_1), \ldots, MS(t_n)$ computed for the n direct subtrees of $a(t_1, \ldots, t_n)$. For symbols a of rank n and trees $t_1, \ldots, t_n$, the value of $\mathcal{F}(MS(t_1), \ldots, MS(t_n))$ is defined to be

$$Cl(Comp_a(\, Filt_{a,1}(MS(t_1))\,, \ldots,\, Filt_{a,n}(MS(t_n))\,))$$

where:

- The $Filt_{a,i}$ are filter functions, filtering items from the respective match sets based e.g. on the values of a and i. Filtering is based on certain elements of children's match sets never contributing to the parent's match set. Such a child match set element may thus be safely disregarded for the computation of the parent's match set. Note that the identity function is among these filter functions.
- The $Comp_a$ are composition functions, which result in those subtrees of RHSs that are compositions of the subnodes' (filtered) match set elements and the symbol a.
- Cl is a closure function, adding e.g. nonterminal LHSs corresponding to complete RHSs that are in the composite match set.

The resulting algorithms are Algorithm (MATCH-SET, REC) (not using filter functions) and Algorithms (MATCH-SET, REC, FILTER) (with different instantiations of filter functions).

As an example of recursive match set computation, assume that we want to compute $MS(a(b(c), d))$ and that we use the identity function as a filter function (i.e. no filtering is applied). Furthermore, assume that $MS(b(c)) = \{b(c), b(B), B\}$

and $MS(d) = \{d, B\}$ have already been computed. Based on this, $MS(a(b(c), d))$ will contain $a(b(c), d)$ and $a(B, d)$ by composition with a, and B and S by the closure function, since $S \Rightarrow a(b(c), d)$ and $B \stackrel{*}{\Rightarrow} S$. No other elements are included in $MS(a(b(c), d))$.

It is straightforward to show that match sets and relations between them, as computed by Algorithm (MATCH-SET, REC) with particular item sets, correspond to states and transition relations of DFRTAs obtained by particular automata constructions as in Section 6. Recursive match set computation and the use of a DFRTA as an acceptor are simply two views on one approach [9, Chapter 5]. This correspondence is indicated by the dotted line in Figure 1.

To improve computation efficiency, values of *MS* cq. the acceptance function of the DFRTA are usually *tabulated* to prevent recomputation. Such tabulation uses a bijection between states (elements of $\mathcal{P}(Items)$) and integers for indexing the tables. The tabulation starts with symbols of rank 0, creating a state for each of them, and continues by computing the composition of symbols with match sets represented by existing states, for as long as new states are encountered, i.e. the computation is performed for the reachable part of state set $\mathcal{P}(Items)$ only. Such reachability-based tabulation is essentially straightforward, but somewhat intricate for trees/n-ary relations, even more so in the presence of filtering. We therefore do not present an example here; see e.g [9, Chapter 5] or [15] instead.

In practice, the size of the RTGs used leads to large but usually sparse tables: e.g for instruction selection, an RTG may well have hundreds of productions and lead to tables of over 100 MB. Filtering is therefore used to reduce storage space. For example, given match set $MS(b(c))$ above, $b(B)$ can be filtered, as it does not occur as a subtree of any *Item* in G. Different item categories can be filtered out (and may lead to different space savings, depending on the grammar):

- Filtering trees not occurring as proper subtrees (such as $b(B)$); filter TFILT, originally by Turner [19].
- Filtering trees not occurring as the ith child tree of a node labeled a; filter CFILT, originally by Chase [4,15].
- One of two new filter functions. Our research in taxonomizing the existing algorithms and filter functions lead us to describe these new ones, which can be seen as simplifications of Chase's filter functions yet somewhat surprisingly had not been described before:
  - Filtering trees based on index i only, i.e. not occurring as the ith child tree of any node; filter IFILT.
  - Filtering trees based on symbol a only, i.e. not occurring as a tree of a node labeled a at any child position; filter SFILT.

Even more surprisingly given their non-appearance in the literature, these two filters turn out to outperform Chase's filter on both text book example RTGs and instruction selection RTGs for e.g. the Intel X86 and Sun SPARC families: the index filter results in lower memory use, while the symbol filter results in slightly faster tabulation time than with Chase's filter. The experimental results have been described in detail in [5,9,18].

8 Algorithms using stringpath matching

The third subgraph (detail SP-MATCHER and below) of the taxonomy in Figure 1 contains algorithms for tree acceptance that are derived from algorithms for tree *pattern matching*. We only briefly sketch the main ideas.

The tree pattern matchers that we use in these tree acceptors reduce tree pattern matching to string pattern matching, using a technique first described in [16]. Each tree can be fully characterized by a set of stringpaths, and a tree pattern matches at a certain position in a tree if and only if all its stringpaths do. By traversing the subject tree and using a multiple string pattern matcher (e.g. [1]), matches of stringpaths can be detected. In [8] (originally presented at this conference as [7]) and [9] we discuss such algorithms in more detail and show that a certain DRFTA construction leads to DRFTAs—i.e. deterministic RF tree automata—that are also usable for stringpath matching. With a little extra bookkeeping, a tree pattern matcher of this kind can be turned into a tree acceptor.

9 Other Parts of the Work

Our work on regular tree algorithms has resulted in two taxonomies and a toolkit of algorithms. In this paper, we have mainly reported on one of the taxonomies, although it was pointed out in Section 3 how similar the tree pattern matching and tree acceptance algorithms and taxonomies are. In this section we present some remarks on the rest of the work. We refer the interested reader to [5,9] for more information.

As mentioned in Section 1, taxonomies form a good starting point for the construction of highly coherent algorithm toolkits. Based on the taxonomies of tree acceptance and tree matching algorithms, such an (experimental) toolkit was developed as part of our research. The toolkit contains most of the concrete algorithms and automata constructions from the taxonomies, as well as a number of fundamental algorithms and data structures—such as alphabets, trees, regular tree grammars, simple grammar transformations—and some extensions of tree acceptance algorithms to tree parsing and rudimentary instruction selection. The design of the toolkit was guided by the two taxonomies: the hierarchy of the taxonomies determines the class and interface hierarchies of the toolkit, and the abstract algorithms lead to straightforward method implementations. The toolkit, called FOREST FIRE, is implemented in Java and accompanied by a graphical user interface (GUI) called FIRE WOOD. This GUI supports input, output, creation and manipulation of data structures from the toolkit and was used to interactively experiment with and get insight into algorithms. More details on the toolkit and GUI can be found in [5,18]. The toolkit and GUI, including source code, example input files and brief manuals, are available for non-commercial purposes via `http://www.fastar.org`.

10 Concluding Remarks

The two taxonomies we constructed cover many algorithms and automata constructions for tree acceptance and tree pattern matching, which appeared in the literature in the past forty years. As for earlier taxonomies, their construction required a lot of time and effort to study original papers and distill the published algorithms' essential details (more so than in usual scientific research, which is typically limited to

studying one or a few existing publications and building on those). Abstraction and sequentially adding details to obtain algorithms were essential and powerful means to clearly describe the algorithms and to make their correctness more apparent.

The uniform presentation in the taxonomies improves accessibility and shows algorithm relations: comparing algorithms previously presented in different styles has become easier and consultation of the original papers is often no longer necessary.

The taxonomies also lead to new and rediscovered algorithms: for example, two new filters were discovered which, though conceptually simple, are practically relevant. Furthermore, Turner's filter was more or less rediscovered. Our initial literature search, although apparently quite extensive, did not find Turner's paper—likely because it was not referred to by any other literature in the same field. As a result, we came up with the rather basic filter independently, before eventually finding it in the literature.

The uniform presentation simplified and guided the high-level design of our toolkit of regular tree algorithms, although the choice of representations for basic data structures still took some time and effort. Experiments with the toolkit provided some interesting results, including the fact that the new filters outperformed Chase's more complex but frequently used filter in many cases.

The results from our research thus are both theoretical and practical, ranging from formal definitions and algorithm taxonomies to a toolkit and experimental results. A form of symbiosis occurred between the theoretical and the practical: the taxonomies were helpful in constructing the toolkit, while the experiments with the toolkit in turn lead to a better understanding of the theoretical definitions and algorithm descriptions, thus helping to simplify the taxonomies.

References

1. A. V. Aho and M. J. Corasick: *Efficient string matching: an aid to bibliographic search.* Communications of the ACM, 18 1975, pp. 333–340.
2. W. S. Brainerd: *Tree generating regular systems.* Information and Control, 14 February 1969, pp. 217–231.
3. M. Broy: *Program construction by transformations: a family tree of sorting programs*, in Computer Program Synthesis Methodologies, A. W. Biermann and G. Guiho, eds., Reidel, 1983, pp. 1–49.
4. D. R. Chase: *An improvement to bottom-up tree pattern matching*, in Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, ACM, 1987, pp. 168–177.
5. L. Cleophas: *Forest FIRE and FIRE Wood: Tools for tree automata and tree algorithms*, in Post-proceedings of the 7th International Workshop on Finite-State Methods and Natural Language Processing (FSMNLP 2008), IOS Press, 2009.
6. L. Cleophas and K. Hemerik: *Forest FIRE: A taxonomy-based toolkit of tree automata and regular tree algorithms*, in Proceedings of the 14th International Conference on Implementation and Application of Automata (CIAA 2009), Springer, 2009.
7. L. Cleophas, K. Hemerik, and G. Zwaan: *A missing link in root-to-frontier tree pattern matching*, in Proceedings of the Prague Stringology Conference (PSC) 2005, August 2005.
8. L. Cleophas, K. Hemerik, and G. Zwaan: *Two related algorithms for root-to-frontier tree pattern matching.* International Journal of Foundations of Computer Science, 17(6) December 2006, pp. 1253–1272.
9. L. G. W. A. Cleophas: *Tree Algorithms: Two Taxonomies and a Toolkit*, PhD thesis, Dept. of Mathematics and Computer Science, Eindhoven University of Technology, April 2008, `http://alexandria.tue.nl/extra2/200810270.pdf`.

10. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi: *Tree automata: Techniques and applications*, 2007, `http://www.grappa.univ-lille3.fr/tata/`.
11. J. Darlington: *A synthesis of several sorting algorithms.* Acta Informatica, 11 1978, pp. 1–30.
12. J. Engelfriet: *Tree Automata and Tree Grammars*, Lecture Notes DAIMI FN-10, Aarhus University, April 1975.
13. C. Ferdinand, H. Seidl, and R. Wilhelm: *Tree automata for code selection.* Acta Informatica, 31 1994, pp. 741–760.
14. F. Gécseg and M. Steinby: *Tree Automata*, Akadémiai Kiadó, Budapest, 1984.
15. C. Hemerik and J. P. Katoen: *Bottom-up tree acceptors.* Science of Computer Programming, 13(1) 1989, pp. 51–72.
16. C. M. Hoffmann and M. J. O'Donnell: *Pattern matching in trees.* Journal of the ACM, 29(1) January 1982, pp. 68–95.
17. H. B. M. Jonkers: *Abstraction, specification and implementation techniques, with an application to garbage collection.* Mathematical Centre Tracts, 166 1983.
18. R. Strolenberg: *ForestFIRE & FIREWood, A Toolkit & GUI for Tree Algorithms*, Master's thesis, Dept. of Mathematics and Computer Science, Eindhoven University of Technology, June 2007, `http://alexandria.tue.nl/extra1/afstversl/wsk-i/strolenberg2007.pdf`.
19. P. K. Turner: *Up-down parsing with prefix grammars.* SIGPLAN Notices, 21(12) December 1986, pp. 167–174.
20. Y. van Dinther: *De systematische afleiding van acceptoren en ontleders voor boomgrammatica's*, Master's thesis, Faculteit Wiskunde en Informatica, Technische Universiteit Eindhoven, August 1987, (In Dutch).
21. B. W. Watson: *Taxonomies and Toolkits of Regular Language Algorithms*, PhD thesis, Dept. of Mathematics and Computing Science, Technische Universiteit Eindhoven, September 1995, `http://www.fastar.org/publications/PhD_Watson.pdf`.
22. B. W. Watson and G. Zwaan: *A taxonomy of sublinear multiple keyword pattern matching algorithms.* Science of Computer Programming, 27(2) 1996, pp. 85–118.
23. R. Wilhelm and D. Mauer: *Compiler Design*, Addison-Wesley, 1995.