ELSEVIER Theoretical Computer Science 242 (2000) 125–142

Theoretical Computer Science

www.elsevier.com/locate/tcs

A new algorithm for linear regular tree pattern matching

Priti Shankar*, Amitranjan Gantait, A.R. Yuvaraj, Maya Madhavan

Department of Computer Science and Automation, Indian Institute of Science, Bangalore 560012, India

Received January 1998; revised July 1998
Communicated by M. Nivat

Abstract

We consider the problem of linear regular tree pattern matching and describe a new solution based on a bottom up technique. Current bottom up techniques preprocess the patterns and construct a finite state tree pattern matching automaton for the purpose. Though matching time is linear in the size of the subject tree, the size of the automaton can be exponential in the sum of the sizes of all patterns. We show here that the problem can be cast as a parsing problem for a context free language, and a solution that uses an extension of the *LR* parsing technique can be devised. Though the size of the resulting pushdown automaton can be exponential in the pattern size in the worst case, there are problem instances for which exponential gains in succinctness of representation are obtained. The technique has been successfully applied to the problem of generation of an instruction selector in a compiler back end. © 2000 Elsevier Science B.V. All rights reserved.

Keywords: Tree pattern matching; Regular tree grammars; Retargetable code generation

1. Introduction

The specification of machine architectures using *Regular Tree Grammars* and *Bottom-Up Rewrite Systems* (BURS) has been extremely useful for the purpose of retargetable code generation [2, 5, 8, 12]. Both specifications have been used to generate finite state tree pattern matching automata, which function as code generators, when augmented with actions that emit code. An input subject tree, (in this application, an intermediate code tree), is traversed by the generated tree automaton, and each time a match of some pattern in the specification is encountered, an action is executed, typically, emission of code. Both, top down [1], and bottom up [2, 8, 12] techniques have been used for preprocessing tree patterns and traversing the subject tree. The basic techniques for the tools constructed have drawn heavily from the seminal papers

* Corresponding author. Tel.: +91 (812) 36441
E-mail address: priti@csa.iisc.ernet.in (P. Shankar).

of Hoffman and O'Donnell [9] and Chase [3], who presented solutions to a related problem, the *Linear Tree Pattern Matching Problem*. Top down techniques are efficient in space, but matching time is nonlinear, the current best space efficient technique having time complexity $O(subsize \times \sqrt{patsize} \times polylog(patsize))$ [4], where *patsize* is the total size of all tree patterns, and *subsize* is the size of the input subject tree. By contrast, bottom up techniques achieve matching in linear time. However, auxiliary space required to store each operator table encoding the finite state tree pattern matching automaton, is exponential in the product of the maximum operator arity, *maxarity*, and *patsize*, each operator requiring one such table. Ferdinand et al. [5] construct a deterministic finite tree automaton encoded as a set of compressed decision trees, along with appropriate index maps. Each decision tree is effectively a compressed version of an operator table. In [2], the approach of [3, 9] has been generalized to handle regular tree patterns augmented with costs. We show that it is possible to solve the regular tree pattern matching problem by constructing a pushdown automaton for the purpose. Though pushdown automata have been used for the problem of table driven code generation [7], the technique proposed there, cannot be applied in general, to the problem of regular tree pattern matching. Our approach preprocesses the patterns using a construction technique that is an extension of that used for *LR* parser construction. We show that there are problem instances for which exponential gains in succinctness of representation are obtained using this technique.

Section 2 presents some background, while Section 3 describes the new algorithm. Section 4 discusses the complexity of the algorithm, and contains results from the application of this algorithm to a test case.

2. Regular tree pattern matching

Let A be a finite alphabet consisting of a set of operators OP and a set of terminals T. Each operator op in OP is associated with an *arity*, $arity(op)$. Elements of T have arity 0. The set $TREES(A)$ consists of all trees with internal nodes labeled with elements of OP, and leaves with labels from T. The number of children of a node labeled op is $arity(op)$. Special symbols called *wildcards* are assumed to have arity 0. If N is a set of wildcards, the set $TREES(A \cup N)$ is the set of all trees with wildcards also allowed as labels of leaves.

Below, we present some definitions drawn mainly from [2, 5].

Definition 2.1. A *regular tree grammar* G is a 4-tuple (N, A, P, S) where

- N is a finite set of *nonterminal* symbols
- $A = T \cup OP$ is a *ranked alphabet*, with the ranking function denoted by arity.
- P is a finite set of *production rules* of the form $X \rightarrow t$, where $X \in N$ and t is an encoding of a tree in $TREES(A \cup N)$.
- S is the *start symbol* of the grammar.

A *tree pattern* is represented by the righthand side of a production of P in the grammar above. A production of P is called a *chain rule*, if it is of the form $A \to B$, where both A and B are nonterminals.

Example 2.1. Below is an example of a regular tree grammar.

$$G = (\{S, R, B\}, \{:=, +, deref, sp, c\}, P, S),$$

where P is given by

$S \to\ := (deref(B), R)$
$S \to\ := (deref(c), R)$
$B \to sp$
$B \to R$
$R \to c$
$R \to B$
$R \to +(B, R)$
$R \to +(R, B)$
$R \to deref(B)$
$R \to +(R, c)$
$R \to +(c, R)$
$R \to +(B, c)$

Derivation sequences are defined in the usual way. However, we note that the objects being derived are trees. An X-derivation tree, D_X, for G has the following properties:
1. The root of the tree has label X.
2. If X is an internal node, then the subtree rooted at X is one of the following three types; (For describing trees we use the usual list notation)
- $X(D_Y)$ if $X \to Y$ is a chain rule and D_Y is a derivation tree rooted at Y.
- $X(a)$ if $X \to a, a \in T$ is a production of P.
- $X(op(D_1, D_2, \ldots, D_k))$ if $X \to op(t_1, t_2, \ldots, t_k)$ is an element of P, $D_i = {}_{X_i}$ if $t_i = X_i \in N$, and $D_i = t_i$ if $t_i \in TREES(A \cup N)$.

The language defined by the grammar is then the set

$$L(G) = \{t \mid t \in TREES(A), \quad \text{and} \quad S \Rightarrow^* t\}.$$

Fig. 1 displays a subject tree in $L(G)$ for the grammar in Example 2.1, and a corresponding S-derivation tree.

The pattern represented by the righthand side of a production used at a particular node, is said to *match* at that node. Thus, for the subject tree of Fig. 1, pattern $:= (deref(c), R)$ matches at node with label $:=$, and pattern $+(Bc)$ matches at node with label $+$. The problems of regular tree pattern matching, and finding S-derivations are therefore equivalent. The matching problem we will address in this paper is: *Given a regular tree grammar, and a subject tree, find a representation of all derivation trees for the subject tree.* In the next section, we describe a solution that constructs a pushdown automaton to solve the problem.

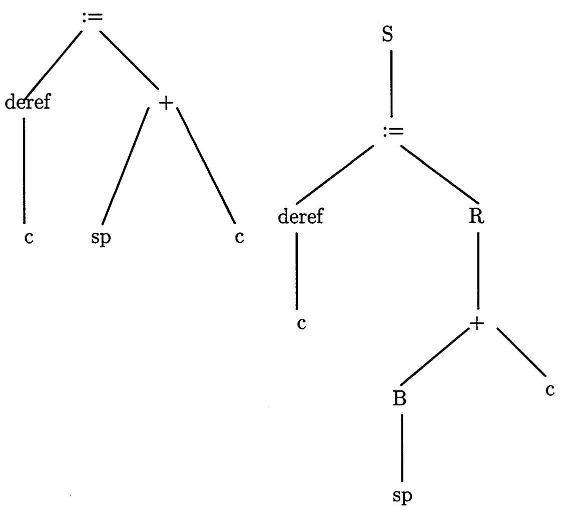

Fig. 1. Tree in the language $L(G)$ for the grammar of Example 2.1.along with its derivation tree.

3. The new technique

The idea of using pushdown automata for table driven code generation had been proposed earlier by Graham and Glanville [7]. However, their approach cannot be applied in general, to the problem of regular tree pattern matching, as it does not carry forward all possible choices in order to be able to report all matches. The technique we describe here can either be viewed as an extension of the $LR(0)$ parsing strategy [6], or as a restriction of the Earley algorithm [11]. We expand on the former viewpoint as it allows for a simple description of the technique.

Let G' be the context free grammar obtained by replacing all righthand sides of productions of G by postorder listings of the corresponding trees in $TREES(A \cup N)$. Note that G is a regular tree grammar whose associated language contains trees, whereas G' is a context free grammar whose language contains strings with symbols from A. Of course, these strings are just the linear encodings of trees.

For purposes of our algorithm, we need grammars in *normal form* [2], defined below.

Definition 3.1. A production is said to be in normal form if it is in one of the three forms below

- $A \rightarrow B_1B_2 \ldots B_k op$ where $A, B_i, i = 1, 2, \ldots, k$ are all nonterminals, and op has arity k.
- $A \rightarrow B$, where A and B are nonterminals.
- $B \rightarrow b$, where b is a terminal.

A grammar is in normal form if all its productions are in normal form. Any grammar can be put into normal form by the introduction of new nonterminals.

Example 3.1. The grammar of Example 2.1 can be converted into a context free grammar in normal form as follows:

$S \to YR :=$
$S \to WR :=$
$Y \to B\ deref$
$W \to C\ deref$
$R \to C$
$B \to SP$
$B \to R$
$R \to B$
$R \to BR+$
$R \to RB+$
$R \to B\ deref$
$R \to RC+$
$R \to CR+$
$R \to BC+$
$C \to c$
$SP \to sp$

Let $post(t)$ denote the postorder listing of the nodes of a tree t. The following theorem has an easy inductive proof which is omitted.

Theorem 3.1. *A tree t is in $L(G)$ if and only if $post(t)$ is in $L(G')$. Also any tree α in $TREES(A \cup N)$ that has an associated S-derivation tree in G has an unique sentential form $post(\alpha)$ of G' associated with it.*

We will show formally that the problem of finding matches at any node of a subject tree t is equivalent to that of parsing the string corresponding to the postorder listing of the nodes of t. Assuming a bottom up parsing strategy is used, parsing corresponds to reducing the string to the start symbol, by a sequence of *shift* and *reduce* moves on the parsing stack, with a match of pattern p being reported at node j whenever a production corresponding to pattern p (i.e. with right-hand side a postorder encoding of p) is used to reduce by at the corresponding position in the string. Thus, in contrast with earlier methods that seek to construct *a tree automaton* to solve the problem, we effectively construct a *deterministic pushdown automaton* for the purpose. We note that the grammar G' is in general ambiguous, and finding all derivation trees in G corresponds to finding all derivations in G'.

Example 3.2. The following are two derivation sequences in G' for the string $w = c\ deref\ sp\ c + :=$ corresponding to the tree in $L(G)$ displayed in Fig. 1.
$S \Rightarrow YR := \Rightarrow YBR + := \Rightarrow YBC + := \Rightarrow YBc + :=$
$\Rightarrow Y\ SPc + := \Rightarrow Y\ sp\ c + := \Rightarrow Bderef\ sp\ c\ + := \Rightarrow R\,deref\ sp\ c\ + :=$
$\Rightarrow C\ deref\ sp\ c + := \Rightarrow c\ deref\ sp\ c + :=$

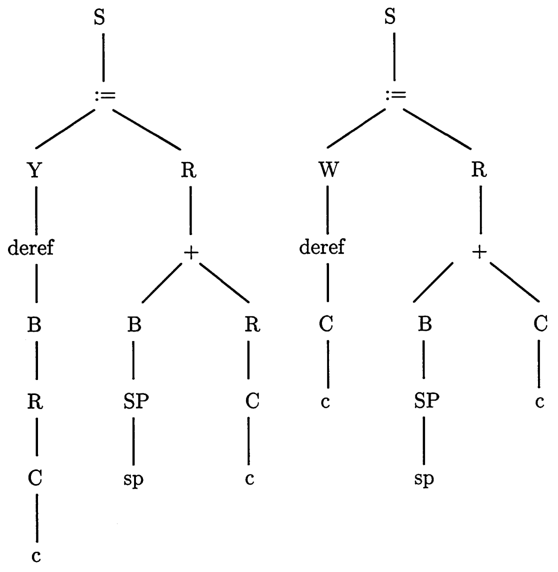

Fig. 2. Two S-derivation trees for the tree with postorder encoding $c\ deref\ sp\ c + :=$.

$$S \Rightarrow WR := \Rightarrow WBC + := \Rightarrow WBc + :=$$
$$\Rightarrow W\ SP\ c + := \Rightarrow Wsp\ c + := \Rightarrow C\ deref\ sp\ c + := \Rightarrow c\ deref\ sp\ c\ + :=$$

The corresponding derivation trees in G are shown in Fig. 2.

3.1. *Extension of the LR(0) parsing algorithm*

We assume that the reader is familiar with the notions of rightmost derivation sequences, handles, viable prefixes of right sentential forms, and items being valid for viable prefixes. Definitions may be found in [10]. By a viable prefix *induced* by an input string we mean the stack contents that result from processing the input string during an LR parsing sequence. If the grammar is ambiguous, then there may be several viable prefixes induced by an input string. The key idea underlying the algorithm is contained in the theorem below:

Theorem 3.2. *Let G' be a normal form context free grammar derived from a regular tree grammar. Then all viable prefixes induced by an input string are of the same length.*

Proof. The proof rests on the following four observations.

1. A shift of any symbol is always followed by a reduction.

2. If the symbol shifted is a terminal symbol, then the length of the viable prefix remains the same, as the handle is of length 1.

3. If the symbol is an operator *op*, then the viable prefix reduces by length *arity*(*op*).
4. Reduction by chain rules does not change the length of the viable prefix.

The first three observations are a consequence of the fact that the grammar is in normal form, and do not depend on the rightmost derivation sequence used. Therefore all parsing sequences for an input string yield viable prefixes of the same length.

In order to apply the algorithm to the problem of tree pattern matching, we need to refine the notion of *matching*, to one of *matching in a left context*.

Definition 3.2. Let $G=(N,T,P,S)$ be a regular tree grammar in normal form, and t be a subject tree. Then pattern β represented by production $X\rightarrow\beta$ matches at node j in left context $\alpha, \alpha\in N^*$ if

1. β matches at node j or equivalently, $X\Rightarrow\beta\Rightarrow^* t'$ where t' is the subtree rooted at j.
2. If α is not ε, then the sequence of maximal complete subtrees of t to the left of j, listed from left to right is $t_1,t_2,\ldots,t_k$, with t_i having an X_i-derivation tree,$1\leqslant i\leqslant k$, where $\alpha=X_1X_2\ldots X_k$.
3. The string $X_1X_2\ldots X_kX$ is a prefix of the postorder listing of some tree in $TREES(A\cup N)$ with an S-derivation.

Example 3.3. Consider the subject tree of Fig. 1. The regular tree grammar corresponding to the context free grammar in Example 3.1 can be easily obtained by converting the righthand sides of all productions into the appropriate form. Pattern $:=(Y,R)$ matches in left context ε. Also pattern c matches in left context YB. This can be deduced by examining the S-derivation tree on the left, in Fig. 2. Also, pattern $+(B,C)$ matches in left context W and $deref(C)$ matches in left context ε. This can be deduced by examining the S-derivation tree on the right, in Fig. 2. The following lemma follows directly from Theorem 3.1. It is useful in proving the theorem that shows that the pattern matching problem can be cast as a parsing problem.

Lemma 3.1. *α is a prefix of a right sentential form of the context free grammar G', if and only if α is a prefix of the postorder listing of a tree in $TREES(A\cup N)$ which has an S-derivation.*

The following theorem forms the basis of our algorithm.

Theorem 3.3. *Let $G=(N,T,P,S)$ be a regular tree grammar, and G' the context free grammar constructed as before. Let t a subject tree with postorder listing $a_1\ldots a_jw, a_i\in A, w\in A^*$. Then pattern β represented by production $X\rightarrow post(\beta)$ of G' matches at node j in left context α if and only if there is a rightmost derivation in the grammar G' of the form*

$$S\Rightarrow^*\alpha Xz\Rightarrow^*\alpha post(\beta)z\Rightarrow^*\alpha a_h\ldots a_jz\Rightarrow^* a_1\ldots a_jz, z\in A^*$$

where $a_h\ldots a_j$ is the subtree rooted at node j.

Proof. (*Only if*) Assume that β matches in left context α. Then clearly, $X \Rightarrow post(\beta) \Rightarrow^* a_h \dots a_j$. Also, if α is not ε, and the subtrees $t_1, \dots, t_k$ are defined as in Definition 3.2, each of the complete subtrees t_i has an X_i-derivation tree, and hence $X_i \Rightarrow^* post(t_i)$. By condition 3 of the definition, αX is a prefix of the postorder listing of a tree that has an S-derivation. From Lemma 3.1, α is a prefix of a right sentential form of G'. Thus $S \Rightarrow^* \alpha X z$ for some $z \in \Sigma^*$, and thus a derivation sequence of the required form exists.

(*If*) Assume that there is a rightmost derivation sequence of the form in the statement of the theorem, and that $\alpha = X_1 X_2 \dots X_k$. Let $t_1, t_2, \dots, t_k$ be the sequence of subtrees whose postorder encodings are derived by $X_1, X_2, \dots, X_k$, respectively. These are all complete subtrees to the left of the subtree rooted at j, and it follows that t_i has an X_i derivation tree, $1 \leqslant i \leqslant k$. Also, $post(\beta)$ can be reduced to X, and hence the pattern β matches at node j. Finally $X_1 X_2 \dots X_l X$ is a viable prefix of G, and by Lemma 3.1, it is the prefix of the postorder listing of a tree that has an S-derivation. Thus pattern β matches in left context α.

Example 3.4. Consider the patterns in Example 3.1. Pattern $:= (Y, R)$ matches in left context ε. This corresponds to the derivation sequence $S \Rightarrow YR := \Rightarrow^* c\ deref\ sp\ c+ :=$. Pattern $C\ deref$ also matches in left context ε. This corresponds to the derivation sequence $S \Rightarrow^* W\ sp\ c+ := \Rightarrow C\ deref\ sp\ c+ := \Rightarrow c\ deref\ sp\ c+ :=$.

The direct correspondence between rightmost derivation sequences in G' matches of regular tree patterns in G, suggests the possibility of using an LR-like parsing strategy for pattern matching. Theorem 3.1 asserts that all viable prefixes are of the same length. This naturally leads to the idea of building a dfa that recognizes *sets* of viable prefixes. We first augment the grammar with the production $Z \rightarrow S\$$ to make it $LR(0)$. We next construct a finite state automaton which we will call the Auxiliary Automaton(AA) as follows:

$$M = (Q, \Sigma, \delta, q_0, F),$$

where each state of Q contains a set of items of the grammar;

$\Sigma = A \cup 2^N$

q_0 is the start state

F is the state containing the item $Z \rightarrow S\$$.

$\delta : Q \times \Sigma \rightarrow Q$

The precomputation of M is similar to the precomputation of the states of the DFA for canonical sets of $LR(0)$ items for a context free grammar. However there is one important difference. In the DFA for $LR(0)$ items, transitions on nonterminals are determined just by looking at the sets of items in any state. Here we have transitions on *sets* of nonterminals. These can not be determined in advance, as we do not know a priori, which patterns are matched simultaneously when matching is begun from a given state. Therefore, transitions on sets of nonterminals are added as and when these sets are determined. Informally, at each step, we compute the set of items generated by making a transition on some element of A. Because the grammar is in normal form, each such transition leads to a state, termed a *matchset* which calls for a reduction by one or more productions. A reduction involves popping off a set of handles from the parsing stack, and making a transition on a set of nonterminals corresponding to the lefthand sides of all productions by which we have performed reductions, from the state (called an *LCset*) that is exposed on stack after popping off the set of handles. This gives us, perhaps, a new state, which is then added to the collection if it is not present.

Two tables encode the automaton. The first, δ_A, encodes the transitions on elements of A. Thus it has, as row indices, the indices of the *LCsets*, and as columns, elements of A. The second, δ_{LC}, encodes the transitions of the automaton on sets of nonterminals. The rows are indexed by *LCsets*, and the columns by indices of sets of nonterminals. The operation of the parser, which is, in fact, the tree pattern matcher, is described below.

Algorithm *TreeMatcher*
Input The input string $w = a_1 a_2 \dots a_n\$$ representing a postorder listing of the nodes of the subject tree, and stored in an array $a[1 \dots n+1]$. Also the AA M for the normal form context free grammar corresponding to the regular tree grammar. Let this have state set $Q = \{q_0, q_1, \dots, q_r\}$ with q_0 being the start state.
Output A list of pairs (i, m) such that pattern p_i matches at node m in left context induced by the postorder listing of the sequence of complete subtrees to the left of m.

```
begin
/* Let stack be the parsing stack, initialized to $q_0$, state be the current state of the
parser also initialized to $q_0$, topstack the state currently on top of the stack, and
push and pop the usual stack operations. Let match(state) be the set of patterns
that are matched in that state */
  for $i = 1$ to $n$ do
      $state := \delta_A(state, a[i])$;
      /* The entry in the table $\delta_A$ directly gives the set $m$ of patterns matched,
      match(state),
      as well as the matching nonterminal set $S_m$.*/
      if $match(state) \neq \phi$ then output($i, match(state)$);
      $pop(stack)$ $arity(a[i])$ times;
      $state := \delta_{LC}(topstack, S_m)$;
      output($i, match(state)$);
      /* these matched patterns correspond to chain rules of the form $A \rightarrow B$ that
      are matched*/
      push(state)
  endfor
end.
```

Clearly, the algorithm is linear in the size of the subject tree. It remains to describe the precomputation of the AA coded by the tables δ_A and δ_{LC}.

3.2. Precomputation of tables

We proceed as follows. The start state of the auxiliary automaton contains the same set of items as would the start state of the dfa for sets of $LR(0)$ items. From each state, say q, identified to be a state of the auxiliary automaton, we find the state entered on a symbol of A, say a. (This depends only on the set of items in the first state.) The second state, say m (which we will refer to as a matchstate), will contain only complete items. We then set $\delta_A(q,a)$ to the pair $(match(m), S_m)$, where $match(m)$ is the set of patterns that match at this point, and S_m is the set of lefthand side nonterminals of the associated productions of the context free grammar. Next we determine all states that have paths of length $arity(a)+1$ to q. We refer to such states as *valid left context* states, for q. These are the states that can be exposed on stack while performing a reduction, after the handle is popped off the stack. If p is such a state then we compute the state r corresponding to the itemset got by making transitions on elements of S_m augmented by all nonterminals that can be reduced to because of chain rules. These new item sets are computed using the usual rules that are used for computing sets of $LR(0)$ items. Finally, the *closure* operation on resulting items completes the new item set associated with r.

To compute states which have paths of a given length to a specified state, it is useful to store a function δ^{-1} which for a given state, returns the set of states which have transitions to this state.

The following function $lc(q,a)$, returns a set of states which have paths of length $arity(a)+1$ to a matchstate q, i.e. are valid left context states for q, given q and a.

```
function $lc(q,a)$ : $set\ of\ states$
    begin
        $lc := q$;
        for $i = 1$ to $arity(a)+1$ do
            $lc := \bigcup_{p \in lc} \delta^{-1}(p)$
    end
```

Before we describe the preprocessing algorithm, we define certain functions that operate on sets of items.

The *goto* operation on a set of items and a symbol is encoded as
$goto(itemset, a) = \{[A \to \alpha a.\beta] \mid [A \to \alpha.a\beta] \in itemset\}$

The *reduction* operation on a set of complete items $itemset_1$ (representing a match state)with respect to another set of items $itemset_2$ (representing a valid left context state), is encoded as the function

```
function $reduction(itemset_2, itemset_1)$
    begin
        $reduction = \phi$
        $reduction := reduction \cup \{[A \to \alpha B.\beta] \mid \exists[B \to \gamma.] \in itemset_1$, and $[A \to \alpha.B\beta]$
            $\in itemset_2\}$
        repeat
            $reduction := reduction \cup \{[A \to B.\beta] \mid \exists[B \to C.] \in reduction$ and
                $[A \to .B\beta] \in itemset_2\}$
        until no change to $reduction$.
    end
```

The *closure* operation on a set of items *itemset* is encoded as follows:

```
function $closure(itemset)$
    begin
        repeat
            $itemset := itemset \cup \{[A \to .\alpha] \mid [B \to .A\beta] \in itemset\}$
        until no change to $itemset$
        $closure := itemset$
    end
```

Define for a pair of item sets $itemset_1$ and $itemset_2$, the function:
$ClosureReduction(itemset_2, itemset_1) = closure(reduction(itemset_2, itemset_1))$

We describe below two preprocessing algorithms. Algorithm *Preprocess* is an exact algorithm which constructs the auxiliary automaton. We note that in algorithm *Preprocess*, each time a transition is added to the automaton, we need to go through a while loop to check whether any new valid left context states are discovered for existing match states, and if so, add new transitions. This process is time consuming. In algorithm *SimplePreprocess*, we check a necessary condition for a state to be a valid left context state for a match state, by just inspecting the item sets in the two states. Thus we do not need the while loop. Simply speaking, we check whether for each complete item $A \to \alpha.$ in the matchstate, there is an item of the form $A \to .\alpha$ in the LCstate, and whether there is a tally of such items. If this is the case then we declare the LCstate to be a valid left context state. More formally, let $rhs(m)$ be the righthandsides of productions corresponding to complete items in a matchstate m.

Define $NTSET(p, rhs(m)) = \{B \mid B \to .\alpha \in itemset(p), \alpha \in rhs(m)\}$. Then a necessary, but not a sufficient condition for p to be a valid left context state for a matchstate corresponding to a matchset m is $NTSET(p, rhs(m)) = S_m$. (The condition is only necessary, because there may be another production that always matches in this left context when the others do, but which is not in the matchset.) We encode the function $validlc(p, m)$ as follows:

```
function $validlc(p, m)$ : $boolean$
    begin
        if $NTSET(p, rhs(m)) = S_m$ then $validlc := true$ else $validlc := false$
    end
```

For both algorithms we maintain a worklist *list*, and a set *lcsets* which finally contains all the *LCsets*.

```
**Algorithm** *Preprocess*
/* We assume that sets of items represent states and vice versa*/
*Input* A context free grammar $G'$ in normal form representing regular tree grammar $G$.
*Output* The auxiliary automaton encoded by tables $\delta_A$ and $\delta_{LC}$.
begin
$lcsets = \phi$;
$matchsets = \phi$;
$list = closure(\{[S \to .\alpha] \mid S \to \alpha \in P\})$;
    while¹ list is not empty do
      delete next element $q$ from list and add it to lcsets
      for¹ each symbol $a \in A$ such that $goto(q,a)$ is not empty do
        $m := goto(q,a)$;
        $\delta^{-1}(m) = \delta^{-1}(m) \cup q$;
        $matchsets := matchsets \cup \{m\}$;
        $\delta_A(q,a) := (match(m), S_m)$;/*$S_m$ is the set of matching nonterminals*/
        for² each state $r$ in $lc(m,a)$ do
            $p := ClosureReduction(r,m)$;
            if $p$ is not in list or lcsets then append $p$ to list;
            $\delta_{LC}(r,S_m) := (match(p), p)$;/*$match(p)$ is the set of matching patterns
            corresponding to chain rules*/
            $\delta^{-1}(p) := \delta^{-1}(p) \cup \{r\}$;
        endfor²
    endfor¹
    $change := true$
    while² $change = true$ do
        $change := false$;
        for³ each state $m$ in matchsets do
            $a :=$ symbol on which $m$ is entered;
            for⁴ each state $r$ in $lc(m,a)$ such that $(r,m)$ has not been processed do
                $p := ClosureReduction(r,m)$;
                $\delta_{LC}(r,S_m) := (match(p), p)$;
                if $p$ is not in list or lcsets then append $p$ to list and set change
                to true;
                if $\delta^{-1}(p)$ does not contain $r$ then add $r$ to $\delta^{-1}(p)$ and set change
                to true
            endfor⁴
        endfor³
      endwhile²
    endwhile¹
end
```

```
Algorithm SimplePreprocess
Input A context free grammar $G' = (N, T, P, S)$ representing a regular tree grammar $G$
in normal form.
Output The auxiliary automaton represented by tables $\delta_A, \delta_{LC}$
begin
lcsets := $\phi$;
matchsets := $\phi$;
list := closure($\{[S \to .\alpha] \mid S \to \alpha \in P\}$);
  while list is not empty do
    delete next element $q$ from list and add it to lcsets;
    for each $a \in A$ such that goto($q, a$) is not empty do
      $m$ := goto($q, a$);
      $\delta_A(q, a) := (match(m), S_m)$;
      if $m$ is not in matchsets then
            matchsets := matchsets $\cup \{m\}$;
            for each state $r$ in lcsets do
              if validlc($r, m$) then
              $p$ := ClosureReduction($r, m$);
              $\delta_{LC}(r, S_m) := (match(p), p)$;
              if $p$ is not in list or lcsets then append $p$ to list endif
              endif
            endfor
      endif
  endfor
  for each state $t$ in matchsets do
      if validlc($q, t$) then
            $s$ := ClosureReduction($q, t$);
            $\delta_{LC}(q, S_t) := (match(s), s)$;
            if $s$ is not in list or lcsets then append $s$ to list endif;
      endif
    endfor
  endwhile
end
```

The algorithms *Preprocess* and *SimplePreprocess* create the tables δ_A and δ_{LC} to be used during matching.

3.3. Correctness of the algorithm

We will now show that the dfa constructed satisfies the property defined below.

Property. $LR(0)$ Item $[A \to \alpha.\beta]$ is valid for viable prefix $\gamma = X_1X_2 \ldots X_k$ in $S_1S_2 \ldots S_k$ if and only if $\delta(q_0, S_1S_2 \ldots S_k)$ contains $[A \to \alpha.\beta]$.

Proof. We give an informal proof that the property is satisfied, based on the proof that a similar property (differing from the one above only in that the function δ is applied on a single viable prefix instead of a set of viable prefixes) holds for a dfa for canonical set of $LR(0)$ items [10]. The construction using algorithm *Preprocess* may be viewed as a subset construction beginning with the $LR(0)$ dfa. We begin with the start state. Suppose m is a matchstate reached during some point in the algorithm, with $S_m = \{X_1, X_2, \ldots, X_m\}$. In the $LR(0)$ dfa we would have individual transitions on $X_1, X_2, \ldots, X_n$ to different states. Here, we collect the items in all those states into a single state, and then augment the items with those obtained with the application of chain rules and got by the closure operation.(This is exactly what *ClosureReduction* does given a matchstate and an LCstate). Thus the state we reach in AA on a label sequence $S_1S_2 \ldots S_n$ is got by merging individual states we would have reached by following individual viable prefixes of the form $X_1X_2 \ldots X_n$ in the dfa for $LR(0)$ item sets induced by the same input string. The property follows from the fact that it holds for each of the viable prefixes in the $LR(0)$ automaton.

3.4. Table compression

It is possible to compress the tables by defining certain equivalence relations on the set *lcsets*.

Let *arityset* be the set of arities of symbols of A. Define a set of equivalence relations $\{R_i \mid i \in arityset\}$ on the set *lcsets* as follows.

If p and q are in *lcsets*, then pR_iq if $\delta_A(p,a) = \delta_A(q,a)$ for all a with $arity(a) = i$. The table δ_A now splits into several tables δ^i_A, one for each arity. The rows of the table δ^i_A correspond to the equivalence classes of R_i. The columns correspond to columns of A_i where A_i is the set of symbols of A with arity i.

Define the set NT_i as

$$NT_i = \{B \mid B \to \alpha a \in P, arity(a) = i\}$$

Equivalence relation U_i is defined on *lcsets* as follows. For states p and q, pU_iq if for all nonterminals B in NT_i, $\delta_{LC}(p,B) = \delta_{LC}(q,B)$. The table δ_{LC} now splits into several tables δ^i_{LC}, one for each value of arity. The table δ^i_{LC} has one row for each equivalence class of U_i and a column for each distinct set of matching nonterminals that is a subset of NT_i.

The compression of the tables can be done on line, while the tables are generated, as the equivalence classes can be computed from the sets of items associated with the states. Each equivalence relation would need an index map which maps from original indices to indices in that equivalence class, which are then used to access table entries. During matching, if the next symbol is in A_i, then table δ^i_A is first consulted through its index map, followed by a lookup of table δ^i_{LC}.

4. Complexity of the algorithm

4.1. Auxiliary space complexity

Let *lcsets* be the set of LC sets, $|R_i|$ $(|U_i|)$ be the number of equivalence classes of $R_i(U_i)$, and $RC_i(UC_i)$ the equivalence classes of $R_i(U_i)$, $i \in arityset$.

The following maps and tables have to be maintained for use during matching, for $i \in arityset$.

- $\delta_A^i : RC_i \times A_i \rightarrow (matchsets \times 2^{NT_i})$
- $\mu_i : lcsets \rightarrow RC_i$
- $\lambda_i : lcsets \rightarrow UC_i$
- $\delta_{LC}^i : UC_i \times 2^{NT_i} \rightarrow lcsets$

It is evident that the sizes of *lcsets* and *matchsets* determine the sizes of the tables, so we first compute loose bounds on the sizes of these sets. Let h be the maximum height of a tree pattern. This, for a regular tree grammar, is the maximum height of trees in *TREES*(A) to be considered using the conventional algorithm, to obtain all the matchsets. Let *nLC* be the size of the set *LCsets*, and *nM* that of *matchsets*.

Lemma 4.1. $nLC + nM \leqslant 2^{|NT| \times (maxarity-1) \times h}$.

Proof. Consider a node in any derivation tree for a subject tree of height bounded by h. The number of maximal complete subtrees to the left of the node represents the length of a path in the auxiliary automaton, encoding a set of viable prefixes induced by the prefix of the input postorder string ending at the postorder predecessor of the terminal or operator associated with the node. The edges of this path are labelled by matching nonterminal sets. The maximum length of any such path is $(maxarity - 1) \times h$. Since each edge on the path may be labeled by a set of matching nonterminals, the a bound on the total number of states of the auxiliary automaton is $2^{|NT| \times (maxarity-1) \times h}$.

A relevant question that arises is: Are there any gains in succinctness of representation that arise from using a pushdown automaton instead of a finite state tree pattern matching automaton? The following theorem answers the question.

Theorem 4.1. *There exist families of languages for which the input patterns have height h, the finite state tree tree pattern matching automaton has number of states* $O(2^{2^h})$, *whereas the auxiliary automaton has size* $O(2^{h+2})$.

Proof. We modify slightly, the family of balanced binary tree patterns described in [9], which constitute a worst case input instance for algorithms that construct bottom up finite state tree pattern matching automata. Let P_j^i, $i \geqslant 0, 1 \leqslant j \leqslant 2^i$ be a class of balanced binary tree patterns of height i with all internal nodes labeled $a \in OP$. There are two nonterminals, S, the start symbol, and V. In P_j^i, all the leaves are labeled V, except the j^{th} leaf from the left which is labeled b. From the set of tree patterns of height

i, we construct a set of tree patterns of height $i+1$ as follows:

$$R_j^{i+1} = op(l_j, P_j^i), \quad l_j \in T, \ 1 \leqslant j \leqslant 2^i.$$

To these patterns, we add the patterns b and c. There will be a total of $2^{i+2}+i$ normalized productions. Example 4.1 below shows the set of unnormalized productions for the case $i=3$. For each subset of patterns, containing distinct elements $\{P_{j_1}^i, P_{j_2}^i, \ldots, P_{j_k}^i\}, j_l \in \{1,2,\ldots,2^i\}$, $i \leqslant l \leqslant k$, there is a subject tree for which exactly this set of patterns matches at the root. This is just the binary tree of height i with a at all internal nodes, a b at each leaf position j_l and c at every other leaf. Thus a finite state tree pattern matching automaton contains at least 2^{2^i} states. However, each pattern P_j^i matches in a different left context and therefore no two or more patterns can appear together in any matchset associated with the LR automaton. The number of matchstates and LCstates are both $O(2^{i+2})$ in this case.

Example 4.1. The regular tree grammar (not in normal form) below, illustrates the case $i=3$. The number of normalized productions is 35.

$S \rightarrow op(l_1, a(a(a(b,V), a(V,V)), a(a(V,V), a(V,V))))$
$S \rightarrow op(l_2, a(a(a(V,b), a(V,V)), a(a(V,V), a(V,V))))$
$S \rightarrow op(l_3, a(a(a(V,V), a(b,V)), a(a(V,V), a(V,V))))$
$S \rightarrow op(l_4, a(a(a(V,V), a(V,b)), a(a(V,V), a(V,V))))$
$S \rightarrow op(l_5, a(a(a(V,V), a(V,V)), a(a(b,V), a(V,V))))$
$S \rightarrow op(l_6, a(a(a(V,V), a(V,V)), a(a(V,b), a(V,V))))$
$S \rightarrow op(l_7, a(a(a(V,V), a(V,V)), a(a(V,V), a(b,V))))$
$S \rightarrow op(l_8, a(a(a(V,V), a(V,V)), a(a(V,V), a(V,b))))$
$V \rightarrow c$
$V \rightarrow b$

4.2. Time complexity

We next estimate the time complexity of our algorithm.

We estimate the complexity of Algorithm *SimplePreprocess* as the auxiliary space bounds estimated above hold for this algorithm as well.

The **while** loop is executed $|lcsets|$ times. Within the **while** loop the first **for** loop is executed $|A|$ times, the complexity of each execution being bounded by $|lcsets| \times |NT| \times$ *patsize* assuming that a *ClosureReduction* operation and computation of *validlc* each take time $|NT|$*patsize* and that all set and table entry operations take constant time. The last **for** loop is executed $|matchsets|$ times for each execution of the **while** loop, the complexity of each execution being bounded by $|NT| \times patsize$. The first **for** loop dominates the execution within the **while**. Hence the overall time complexity is bounded by $|lcsets|^2 \times |A| \times |NT| \times patsize$.

4.3. Discussion

A comparison of the space requirements of the scheme based on the *LR* parsing technique, with those of conventional techniques, suggests that the former is likely to have advantages in cases where operators have large arities and when the transition function of the tree automaton is not total. Though Lemma 4.1 indicates that the size of the *LR* tables can be polynomial (of degree h) in the maximum size of an operator table, it is not clear at this point as to what kinds of input instances elicit worst case behaviour. Experimentation with some actual problem instances is necessary to examine the efficiency of this algorithm. Ferdinand et al. [5] point out that in many cases, tables generated by the technique of Chase do not fit into main memory, and propose storing them as decision tables. Such compression techniques result in considerable savings in space when tables are sparse. However, at matching time, a linear number of extra array accesses are needed for each dimension. The *LR* based scheme requires a constant number of array accesses per node.

Our bottom up tree pattern matcher, performs pattern matching by visiting nodes of the subject tree in postorder. The *LR* based strategy has the property that if the subject tree is not in the language defined by the regular tree grammar, then it will detect so at the first symbol at which the postorder listing ceases to be a prefix of a valid listing. (This problem is not encountered in code generation applications, as the input tree is assumed to be in the language.) This is called the *valid prefix property*. Bottom up parsers based on finite tree automata do not have this property. For the problem of simple tree pattern matching, all subject trees are in the language generated by the grammar, so this problem does not arise.

We have run this algorithm on a specification consisting of the tree patterns encoding the instruction set of the MC68030 machine(without costs) [6]. Both *Preprocess* and *SimplePreprocess* were run on the input, the latter producing just one extra state.
Details are given below:
Number of normalized productions: 176, Number of nonterminals: 32
Number of operators of arity 1: 12, Number of operators of arity 2: 15, Number of symbols of arity 0: 19
Size of *LCsets*: 348, Size of *matchsets*: 144
Total table size: 5.6 K

The same problem instance was run through an algorithm that constructs a finite state tree pattern matching automaton using the technique of Chase [13], with states augmented with costs, setting all costs to 0. The total table size including index maps was around 7.7 K. Thus for this instance, the sizes seem to be comparable. Preprocessing times were also comparable. More experimentation is necessary to check the efficiency of this technique, both for precomputation as well as matching.

It appears possible to modify the preprocessor to work in an incremental manner, with respect to pattern additions and deletions. Also states could be augmented with cost information at code generator construction time, so that locally optimal code can be generated, as is the case for tree automata augmented with costs. Future work on application to code generation will concentrate on these problems.

Finally, it must be pointed out that the algorithm suggested in this paper performs a slightly refined version of pattern matching, in that some contextual information can be carried along. This might be useful in applications that require tree matches in specified contexts. Different kinds of linearization strategies for the subject tree may be employed, depending on the type of contextual information to be stored, the main strategy requiring very little modification.

References

[1] A.V. Aho, M. Ganapathi, Efficient tree pattern matching: an aid to code generation, in: Proc. 12th ACM Symp. on Priciples of Programming Languages, 1985, pp. 334–340.
[2] A. Balachandran, D.M. Dhamdhere, S. Biswas, Efficient retargetable code generation using bottom up tree pattern matching, Comput. Lang. 3 (15) (1990) 127–140.
[3] D. Chase, An improvement to bottom up tree pattern matching, in: Proc. 14th Ann. ACM Symp. on Principles of Programming Languages, 1987, pp. 168–177.
[4] M. Dubiner, Z. Galil, E. Magen, Faster tree pattern matching, in: Proc. 31st IEEE FOCS '90, 1990.
[5] C. Ferdinand, H. Seidl,, R.Wilhelm, Tree automata for code selection, Acta Inform. 31 (1994) 741–760.
[6] A. Gantait, A new algorithm for tree pattern matching and its application to retargetable code generation, M.E. Project Report, Department of Computer Science and Automation, Indian Institute of Science, Bangalore, December 1996.
[7] R.S. Glanville, S.L. Graham, A new method for compiler code generation, in: Proc. 5th Ann. ACM Symp. on Principles of Programming Languages, 1978, pp. 231–240.
[8] P.Hatcher, T. Christopher, High-quality code generation via bottom-up tree pattern matching, in: Proc. 13th ACM Symp. on Principles of Programming Languages, 1986, pp. 119–130.
[9] C. Hoffmann, M.J. O'Donnell, Pattern matching in trees, J. ACM 29 (1) (1982) 68–95.
[10] J. Hopcroft, J.D. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, Reading, MA, 1979.
[11] Maya Madhavan, Optimal regular tree pattern matching using pushdown automata, MSc(Engineering) Thesis, Dept. of Computer Science and Automation. Indian Institute of Science, Bangalore, 1998.
[12] E. Pelegri Llopart, S. Graham, Optimal code generation for expression trees: An application of BURS theory, in: Proc. 15th Annual ACM Symp. on Principles of Programming Languages, 1988, pp. 294–308.
[13] S. Ravi Kumar, Retargettable code generation using bottom up tree pattern matching, M.E Project Report, Department of Computer Science and Automation, Indian Institute of Science, Bangalore, 1992.