PAPER

An Analysis for Fast Construction of States in the Bottom-Up Tree Pattern Matching Scheme

Kyung-Woo KANG[†], Kwang-Moo CHOE[††], *and* Min-Soo JUNG[†††], *Nonmembers*

SUMMARY In this paper, we propose an efficient method of constructing states in bottom-up tree pattern matching with dynamic programming technique for optimal code generation. This method can be derived from precomputing the analysis which is needed for constructing states. The proposed scheme is more efficient than other scheme because we can avoid unfruitful tests in constructing states at compile time. Furthermore, the relevant analyses needed for this proposal are largely achieved at compile-compile time, which secures actual efficiency at compile time.

key words: *compilers, code generator generator, tree grammar, dynamic programming*

1. Introduction

Many code generators (CG) accept an intermediate representation (IR) in a tree structure and convert it to an equivalent target program. The tree structure is proper for representing semantic of source program efficiently and its manipulation is considered easy [15]. Since the development of bottom-up tree pattern matching by Hoffman and O'Donnell [5], the bottom-up tree pattern matching has been accepted as a practical technique for CG and a code generator generator (CGG) [1], [2], [4], [6], [7], [11], [14], [15]. The specification of the CGG which is actually the machine specification contains the tree grammar which consists of rules. Each rule has a cost associated with its semantic action.

The bottom-up tree pattern matching scheme is much faster than any other tree pattern matching scheme theoretically [5], [12]. The Bottom-Up Rewrite System (BURS) theory is efficient because Dynamic Programming (DP) can be done at compile-compile time [11], [15]. However, BURS has the restriction that the costs used in the tree grammar must be constant. The bottom-up tree pattern matching scheme adapting DP at compile time allows the arbitrary cost values [6], which admit a larger class of tree grammars [11] but may cause inefficiency at compile time. This paper describes a program that reads a machine specification and writes a bottom-up tree pattern matching scheme that does DP at compile time.

We emphasize the efficiency of the bottom-up tree pattern matching scheme which allows the arbitrary cost values. The bottom-up tree pattern matching scheme with DP [6] traverses the IR tree twice. In the first traversal, the scheme computes a state at every node of the IR tree in a bottom-up direction. A state can be extracted along the sequence of the rules. In the second traversal, the scheme will find the least-cost cover in a top-down direction. Then a target code is produced by executing the semantic actions for the rules used for the least-cost cover. Previous scheme [6] computes a state using all rules in the given sequence. However, we can infer that only reduced number of rules can be used in computing states, which is our primary intention. To implement our intention, we firstly transform the sequence of rules into several sets called match set and the match set transition tables. A match set is a set of patterns which must be used for construction of a state. The state of a node can be computed using a match set which is determined from the match sets of its child nodes using the transition tables. In this paper, the transition tables are hard coded into code generator which uses bottom-up tree pattern matching scheme with dynamic programming.

In Sect. 2, definitions and background are introduced. In Sect. 3, we propose an efficient method of constructing states. In Sect. 4, we show experimental results for the scheme. A summary and the concluding remarks are given in Sect. 5.

2. Definitions and Background

We describe a representing scheme for bottom-up tree pattern matching with DP, upon which our study is based. Moreover, necessary definitions for presenting proposals in Sect. 3 will be described.

An *alphabet* (written as Σ) is a finite set of operators denoted as a, b, $c, \ldots$. Each operator has a fixed arity (written as $arity(a) \geq 0$). We write Σ_n for $\{a \mid arity(a) = n\}$. The *tree language* over Σ (written as T_Σ) is defined as follows:

- a is a *tree* in T_Σ, if $a \in \Sigma_0$.
- $a(t_1, \ldots, t_n)$ is a *tree* in T_Σ, if $a \in \Sigma_n$ and $t_1, \ldots, t_n$

Manuscript revised June 17, 1997.

[†]The author is with the Supercomputer Center, Electronic & Telecommunications Research Institute, Taejon, Korea.

[††]The author is with the Faculty of the Department of Computer Science, Korea Advanced Institute of Science and Technology, Taejon, Korea.

[†††]The author is with the Faculty of the Department of Computer Engineering, KyungNam University, Masan, Korea.

are *trees* in T_Σ.

A subtree of a tree is identified through its *position* which is a sequence of relative integers from the root of the tree. *Position* of a node of a tree $t \in T_\Sigma$ is defined by a sequence of relative integers from the root to the node. If $t = a(t_1, \ldots, t_n)$, then $Pos(t) = \{\varepsilon\} \cup \{i \cdot p' \mid 1 \leq i \leq n, p' \in Pos(t_i)\}$. If $p \in Pos(t)$, then the subtree t/p of t is defined as follows:

- $t/p = t$, if $p = \varepsilon$ and
- $t/p = t_i/p'$, if $t = a(t_1, \ldots, t_n)$ and $p = i \cdot p'$.

The set of positions $Pos(t)$ of a tree t is partially ordered by $\preceq$; For $p_1, p_2 \in Pos(t)$ such that $p_1 \neq \varepsilon$ and $p_2 \neq \varepsilon$, $p_1 \preceq p_2 \iff i < j$ or $i = j, p'_1 \preceq p'_2$ where $p_1 = i \cdot p'_1$, $p_2 = j \cdot p'_2$.

A set V is a countably infinite set of *variables* denoted as $v_1, v_1, v_3, \ldots$. Each variable is a symbol of arity zero to be replaced by a tree $t \in T_{\Sigma\cup V}$. The *pattern* is a tree in $T_{\Sigma\cup V}$. When there are no constraints between values used to replace any two variables, the pattern is called linear.

A *substitution* is a map $\Theta : V \to T_{\Sigma\cup V}$ which can be extended to all trees by defining $a(t_1, \ldots, t_n)\Theta = a(t_1\Theta, \ldots, t_n\Theta)$ for every n-ary operator a $(n > 0)$. $t\Theta$ is also written as $t[t_1\backslash v_1, \ldots, t_n\backslash v_n]$ if the set of the variables occurring in t is $\{v_1, \ldots, v_n\}$ and $v_i\Theta = t_i$ for all i. Assume that α is a pattern in $T_{\Sigma\cup V}$. α *matches* a tree t if there exists a substitution Θ such that $\alpha\Theta = t$.

Definition 1: A *tree grammar* is a quadruple $G = (N, \Sigma, R, S)$ where

- N is a finite set of *nonterminals* denoted as $A, B, \ldots$.
- Σ is the alphabet of *terminals* denoted as $a, b, \ldots$.
- R is a finite set of *rules* of the form $A \to \alpha$ with α in $T_{\Sigma\cup N}$ and A in N. α, A are also called *pattern*. Each rule has an associated *cost* such as $c(A \to \alpha)$.
- S is a special nonterminal and it represents *start symbol*.

□

A rule $r : A \to \alpha$ is of *type* $(A_1, \ldots, A_n) \to A$ if the i-th nonterminal in α is A_i [10]. For a left side α of the rule r, the pattern $\tilde{\alpha}$ is defined as a pattern in $T_{\Sigma\cup\{v_1,\ldots,v_n\}}$ such that $\tilde{\alpha}$ is obtained from α by replacing, for $1 \leq j \leq n$, the j-th nonterminal by a variable v_j [10]. For a tree grammar G, we define *a-rules* as the set of rules such that *a-rules* $= \{A \to a(t_1, \ldots, t_n) \in R \mid a \in \Sigma_n$ and for $1 \leq i \leq n$, $t_i \in T_{\Sigma\cup N}\}$. A rule of the form $A \to \alpha$ is called *normal form* if α is a pattern of the form either $X \in \Sigma_0 \cup N$ or $a(A_1, \ldots, A_n)$ where $a \in \Sigma_n$ and $A_1, \ldots, A_n \in N$. A tree grammar is called *normal form* if all rules are in normal form. Any tree grammar can be converted into *normal form* tree grammar by introducing several *nonterminals* and *rules* [1]. In this paper, the tree grammar is assumed to be a normal form.

A *cover* of $t \in T_{\Sigma\cup N}$ to A is a sequence of pairs $\langle rule, position\rangle \in R \times Pos(t)$. If τ is a cover of t/p to X, then τ satisfy the following conditions:

- If $\tau = \varepsilon$ then $t = X$
- If $\tau = \tau_1 \cdots \tau_n\langle r, p\rangle$ for some rule $r : A \to \alpha \in R$ of type $(X_1, \cdots, X_n) \to A$, then $t/p = \tilde{\alpha}[t_1\backslash v_1, \cdots, t_n\backslash v_n]$ and τ_i is a cover of t_i to X_i for $1 \leq i \leq n$.

The *last rule*(written as $last(\tau)$) of a cover τ is defined as follows. If $\tau = \langle r_1, p_1\rangle \cdots \langle r_n, p_n\rangle$ and $r_i \in R$ for $1 \leq i \leq n$ then $last(\tau) = r_n$. The *tree language* of G relative to α (written as $L(G, \alpha)$) is:

- $\{t \in T_\Sigma \mid \exists\tau : \tau$ is a cover of t to $A\}$ if $\alpha = A \in N$.
- $\{t \in T_\Sigma \mid t = \tilde{\alpha}[t_1\backslash v_1, \cdots, t_n\backslash v_n]$ and $\exists\tau_i : \tau_i$ is a cover of t_i to A_i for $1 \leq i \leq n\}$ if $\alpha = a(A_1, \cdots, A_n)$.

The *cost* associated with a cover τ is the sum of the costs associated with each *rule* in the cover (written as $C(\tau)$ which is the extension of cost of a rule):

- If $\tau = \varepsilon$ then $C(\tau) = 0$.
- If $\tau = \langle r_1, p_1\rangle \cdots \langle r_n, p_n\rangle$ then $C(\tau) = \sum_{i=1}^{n} C(r_i)$.

In *covers* of t to A, one cover with the minimum cost is called *least-cost cover* of t to A (written as $LCV(t, A)$). Then the goal of the tree pattern matching scheme is to find the $LCV(t, S)$. Evaluation procedure of $LCV(t, A)$ consists of two phases. The first phase annotates a state on each node of the IR tree t in a bottom-up way. A state is a set of triples (nonterminal, rule, cost) which is called an *item*. In an item, rule and cost are computed for nonterminal [6]. The state annotated on the root node of $t = a(t_1, \ldots, t_n)$ is $\{(B, cost(t, B), rule(t, B)) \mid B \in N\}$ where

- $cost(t, B) = min\{\sum_{i=1}^{n} c_i + C(r) + C(\tau) \mid r : B' \to \alpha \in$ *a-rules* and if r is type $(B_1, \ldots, B_n) \to B'$, then c_i is $cost(t_i, B_i)$ for $1 \leq i \leq n$ and τ is a cover of B' to $B\}$.
- $rule(t, B) = last(\tau)$ such that τ is a *cover* of t to B and $C(\tau) = cost(t, B)$.

In the second phase, the scheme finds the least-cost cover of t to A while traversing the IR tree in a top-down direction. If s is the state annotated on the root node of t/p, then $LCV(t, B) = LCV(t_1, B_1) \cdots LCV(t_n, B_n) \cdot \langle r, p\rangle$ where $(B, c, r{:}B \to \alpha) \in s$, r is of type $(B_1, \ldots, B_n) \to B$ and $t = \tilde{\alpha}[t_1\backslash v_1, \ldots, t_n\backslash v_n]$.

3. An Efficient Method of Constructing States

In the previous work of [6], the cost ($cost(t, A)$) is evaluated from a sequence of all rules in *a-rules* when computing an item in the state. We can infer that only reduced number of rules can be used in computing states.

Table 1 Numbers of tree patterns to be checked.

	x86		m68k		sparc		mips	
Programs	iburg	ours	iburg	ours	iburg	ours	iburg	ours
array.c	1582	841	1339	791	755	642	633	529
cf.c	609	379	617	410	375	320	321	270
cq.c	93008	48336	102616	64587	57484	48477	44654	36479
fields.c	1318	701	1098	758	633	532	580	480
sort.c	1180	655	1344	751	679	563	574	469
struct.c	968	495	959	620	842	727	479	399
switch.c	2650	1461	2558	1555	1486	1312	1346	1178
front.c	180	104	231	134	88	78	78	68

Table 2 Numbers of tree patterns to be checked.

	x86		m68k		sparc		mips	
	iburg	ours	iburg	ours	iburg	ours	iburg	ours
Time(sec)	9.1	8.8	9.3	9.1	8.2	8.1	7.5	7.5

This is the point we are claiming in this paper, and we will propose an efficient method of constructing states.

The *match set* of a tree t is a set of the patterns α such that ($A \to \alpha \in R$ or $\alpha \in N$) and $t \in L(G, \alpha)$. If m is the match set of t, then m is calculated as follows: $m = \{\alpha \mid r : A \to \alpha \in R$ and if r is type $(A_1, \ldots, A_n) \to A$ then $t = \tilde{\alpha}[t_1 \backslash v_1, \ldots, t_n \backslash v_n]$, $\exists \tau_i : \tau_i$ is a cover of t_i to A_i for $1 \le i \le n\} \cup \{B \mid \exists \tau : \tau$ is a cover of α to B for $\alpha \in m\}$ (*cf* [5]).

All possible match sets ($Q = \{m \mid t \in T_\Sigma, m$ is the match set of $t\}$) can be computed at compile-compile time. The transition tables among *match sets* can be computed at compile-compile time. These transition tables are used for the efficiency of construction of states. The transition tables are defined as follows: $\delta_a : (Q_N)^n \to Q$, $\mu_a : Q \times [1, n] \to Q_N$ where $Q_N = \{N \cap m \mid m \in Q\}$ and $a \in \Sigma_n$. We assume that $t = a(t_1, \ldots, t_n)$, m is the match set of t and m_i is the match set of t_i for $1 \le i \le n$. Let $q_i = m_i \cap \{A_i \mid A \to a(A_1, \ldots, A_i, \ldots, A_n) \in a\text{-}rules\}$ for $1 \le i \le n$. We define $\mu_a(m_i, i) = q_i$ and $\delta_a(q_1, \ldots, q_n) = m$.

Algorithm 1: Let $G = (N, \Sigma, P, S)$ be a tree grammar. The sets Q, δ_a and μ_a are iteratively determined by $Q = \bigcup_{0 \le j} Q^{(j)}$, $\delta_a = \bigcup_{0 \le j} \delta_a^{(j)}$ and $\mu_a = \bigcup_{0 \le j} \mu_a^{(j)}$ where

1. $Q^{(0)} = \emptyset$, $\delta_a^{(0)} = \emptyset$ and $\mu_a^{(0)} = \emptyset$ for all $a \in \Sigma$;
2. Assume $j > 0$. For $a \in \Sigma_n$ and $m_1, \ldots, m_n \in Q^{(j-1)}$ such that $m_i \cap \{A_i \mid A \to a(A_1, \ldots, A_i, \ldots, A_n) \in a\text{-}rules\} \neq \emptyset$. Let $q_i = m_i \cap \{A_i \mid A \to a(A_1, \ldots, A_i, \ldots, A_n) \in a\text{-}rules\}$ for $1 \le i \le n$. Let $m = \{\alpha \mid$ For each $r : A \to \alpha \in R$ of type $(A_1, \ldots, A_n) \to A$ such that $r \in a\text{-}rules$, $A_i \in q_i$ for $1 \le i \le n\} \cup \{B \mid \exists \tau : \tau$ is a cover of α to B for $\alpha \in q\}$. If $m \neq \emptyset$ then $m \in Q^{(j)}$ and $\delta_a^{(j)}(q_1, \ldots, q_n) = m$ and $\mu_a^{(j)}(m_i, i) = q_i$.

Since $Q^{(j)} \subseteq Q^{(j+1)}$ for $0 \le j$, iteration can be terminated as soon as no new states are generated. Therefore $Q = Q^{(j)}$, $\delta_a = \delta_a^{(j)}$ and $\mu_a = \mu_a^{(j)}$ for the first j with $Q^{(j)} = Q^{(j+1)}$. □

In this paper, it is a primary intention that we use the set Q, δ_a and μ_a to evaluate $cost(t, A)$ of state efficiently.

Theorem 1: Assume that the state annotated on the root node of $t = a(t_1, \ldots, t_n)$ is $\{(A, cost(t, A), rule(t, A)) \mid A \in N\}$, m is the match set of t and m_i is the match set of t_i for $1 \le i \le n$. If $\mu_a(m_i, i) = q_i$ for $1 \le i \le n$ and $\delta_a(q_1, \ldots, q_n) = m$, then $cost(t, A) = min\{\sum_{i=1}^{n} c_i + C(r) + C(\tau) \mid r : B \to \alpha \in a\text{-}rules$, $\alpha \in m$ and if r is type $(A_1, \ldots, A_n) \to B$, then c_i is $cost(t_i, A_i)$ for $1 \le i \le n$ and τ is a cover of B to $A\}$. □

Also we describe another speed-up technique which can be applied when there is only one pattern in a match set to compute the state. From the definition of the state, it is required that each nonterminal has only one cost, and its value is relative to that of the other nonterminals in the match set. When one pattern of the match set is α, the evaluation of $cost(t, A)$ is simplified as follows: $cost(t, A) = min\{C(r) + C(\tau) \mid r : B \to \alpha \in R$ and τ is a cover of B to $A\}$.

4. Experimental Results

Based on the Algorithm 1, the CGG proposed in this paper is implemented for experiment and comparison with the related work **iburg**. Our CGG is a modified version of **iburg** which produces a CG of **lcc** [4]. **iburg** is a recently developed CGG adopting the bottom-up tree pattern matching with DP technique at compile time. We show improvements in compile time by the experiment on MC68000, x86, mips and sparc CGs. Relevant statistics for the CGs are shown in Table 1. The standard of the comparison is the number of tree patterns checked in computing the state at each node. The C programs which were tested are the test suites of **lcc**. Table 2 shows amount of time required in compiling C program (5317 lines) by two versions of **lcc** (original version and our version). Those amounts are required in compilation only; times spent in preprocessing, assembly, and linking time are excluded. Test compilations were executed on on the Sparc2/40 station with 48 MB under SunOS Release 4.1.2. The time in Table 2 means the lowest elapsed time in seconds chosen among the results of several experimentations on a lightly loaded machine (i.e., (user + system)/elapsed ≥ 0.95).

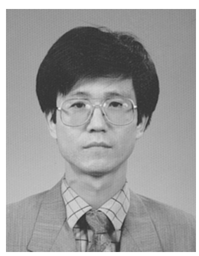
5. Concluding Remarks

In this paper, we proposed an efficient method of constructing states in bottom-up tree pattern matching with DP in the CG. The task of computing a state is divided into two parts; the computation of the match set, the computation of the state. Although the state cannot be computed at compile-compile time because of the evaluation of costs, the match sets can be computed at compile-compile time. We transform the sequence of rules into several match sets and several transition tables. If the match sets and the transition tables is used in constructing the states, then the pattern matcher can avoid about 40% unfruitful tests. However, the size of matcher is larger because the match sets may have common rules. In Addition, we would like to point out that some part of the analyses required in our method can be applied at compile-compile time, which may secure practical efficiency at compile time.

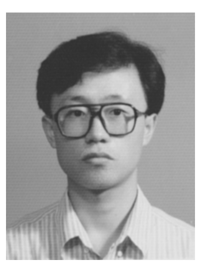
References

[1] A. Balachandran, D.M. Dhamdhere, and S. Biswas, "Efficient retargetable code generation using bottom-up tree pattern matching," Comp. Lang. vol.3, pp.127–140, 1990.
[2] A.V. Aho, M. Ganapathi, and S.W. K. Tjiang, "Code generation using tree matching and dynamic programming," ACM Trans. Prog. Lang. and Syst., vol.1, pp.159–175, 1989.
[3] A.V. Aho, R. Sethi, and J.D. Ullman, "Compilers—Principles, Techniques, and Tools," Addison Wesley, 1986.
[4] C.W. Fraser and D.R. Hanson, "A Retargetable C Compiler: Design and Implementation," The Benjamin/Cummings, 1995.
[5] C.M. Hoffmann and M.J. O'Donnell, "Pattern matching in trees," ACM Journal, vol.1, pp.68–95, 1982.
[6] C.W. Fraser, D.R. Hanson, and T.A. Proebsting, "Engineering a simple, efficient code generator generator," ACM Lett. Prog. Lang. and Syst., vol.6, pp.331–340, 1992.
[7] C.W. Fraser, R.R. Henry, and T.A. Proebsting, "BURG-fast optimal instruction selection and tree parsing," ACM SIGPLAN Notices, vol.4, pp.68–76, April 1991.
[8] D. Comer and R. Sethi, "The complexity of trie index construction," Journal of the ACM., vol.24, pp.428–440, July 1977.
[9] D.R. Chase, "An improvement to bottom-up tree pattern matching," 14th Annual symp. on POPL, pp.168–177, 1987.
[10] C. Ferdinand, H. Seidl, and R. Wilhelm, "Tree automata for code selection," Acta Informatica, vol.31, pp.741–760, 1994.
[11] E. Pelegri-Liopart, "Rewrite systems, pattern matching, and code generation," Ph.D. Dissertation, Report no.UCB/CSD 84/184, CSD, EECS, UCB, CA, May 1988.
[12] E. Pelegri-Liopart and S.L. Graham, "Optimal code generation for expression trees: An application of BURS theory," 15th Annual ACM SIGACT-SIGPLAN symp. on POPL, pp.294–307, San Diego, California, Jan. 1988.
[13] K.-W. Kang and K.-M. Choe, "On the automatic generation of instruction selector using bottom-up tree pattern matching," Tech. Rept. CS-TR-95-93, Dept. of Computer Science, KAIST, 1995.
[14] T.A. Proebsting, "Simple and efficient BURS table generation," SIGPLAN Notices, vol.6, pp.331–340, 1992.
[15] T.A. Proebsting and C. Fischer, "Code generation techniques," Ph.D. Dissertation, Department of Computer Science, University of Wisconsin-Madison, 1992.

Kyung-Woo Kang received the B.S. degree in computer science and statistics from the Kyungsung University, Korea, in 1990, and the M.S. degree in computer science from the Korea Advanced Institute of Science and Technology, in 1992, and the Ph.D. degree in computer science from the Korea Advanced Institute of Science and Technology, Korea, in 1998. Since 1998 he has been a researcher member of the Supercomputer Center, Electronic & Telecommunications Research Institute. His research interests include parsing theory, retargetable compiler, and meta-computing system.



Kwang-Moo Choe received the B.S. degree in electronic engineering from the Seoul National University, Korea, in 1976, the M.S. degree in computer science from the Korea Advanced Institute of Science and Technology, in 1978, and the Ph.D. degree in computer science from the Korea Advanced Institute of Science and Technology, Korea, in 1984. From 1985 to 1986, he was with the AT&T Bell Labs, as a Member of Technical Staff. Since 1984 he has been a faculty member of the Department of Computer Science, Korea Advanced Institute of Science and Technology. He is also the Department chair of the Department of Computer Science. His research interests include formal language theory, parallel evaluation of Logic programs, optimizing compilers, retargetable compiler, and parallelizing compiler.



Min-Soo Jung received the B.S. degree in computer engineering from the Seoul National University, Korea, in 1986, the M.S. degree in computer science from the Korea Advanced Institute of Science and Technology, in 1988, and the Ph.D. degree in computer science from the Korea Advanced Institute of Science and Technology, Korea, in 1994. Since 1990 he has been a faculty member of the Department of Computer Engineering, Kyung-Nam University, Masan, Korea. His research interests include formal language theory, error recovery, and retargetable compiler.