PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT. © ICOT, 1988

PARALLEL COMPUTATIONAL COMPLEXITY OF LOGIC PROGRAMS AND ALTERNATING TURING MACHINES

Yasuo OKABE and Shuzo YAJIMA

Department of Information Science
Faculty of Engineering, Kyoto University
Kyoto 606, JAPAN

ABSTRACT

We argue on the computational complexity of logic programs introduced by Shapiro, and show a close relationship between logic programs and tree-size bounded alternating Turing machines. First, we propose a modified definition of goal-size of logic programs to cope with sublinear space complexity. Second, we present algorithms for simulating logic programs and indexing alternating Turing machines each other. As a result, simultaneous goal-size and depth of logic programs are closely related to space and tree-size of indexing alternating Turing machines respectively. Further, we investigate and characterize many well-known complexity classes like *LOGCFL*, *NC*, *PTIME* and *NPTIME*, via logic programs. In particular, it is shown that context-free languages can be recognized by logic programs with depth $O(\log n)$ and goal-size $O(\log n)$ simultaneously.

1 INTRODUCTIONS

Logic programming languages have lately attracted considerable attention as languages for "fifth generation computers". In particular, since the execution of a logic program can be regard as parallel computation, several parallel logic programming languages such as Concurrent Prolog (Shapiro 1983) and GHC (Ueda 1985) are proposed in order to describe parallel processing. Moreover, many parallel computer architectures for processing logic programs are being developed, and some of them have been actually implemented. It becomes important to give a mathematical foundation to such a parallelism in logic programs, and to relate it to the conventional theory of parallel computation.

Shapiro (1984) showed that there is a close relationship between logic programs and alternating Turing machines (ATMs) (Chandra et al. 1981). In his formulation, a logic programs is regarded as an ATM, which is given a goal as an input string. The derivation of the program corresponds to the computation tree of the ATM, and the ATM accepts the input if and only if the program can solve the goal. He introduced three complexity measures for logic programs, namely, depth complexity, goal-size complexity and length complexity, and showed that these and tree-size respectively by simulating on-line alternating Turing machines and logic programs each other.

His result tells us many things about relations among logic programs and theoretical parallel computation models such as PRAMs and combinational circuits. According to his definition, however, goal-size complexity should be $\Omega(n)$ for the input length n; this is because "input" and "work space" are not distinguished. Corresponding alternating space complexity is also at least linear. For this reason, practically important classes like *PTIME* cannot be characterized via logic programs. Stepanek and Stepankova (1986) gave simulations between off-line alternating Turing machines and logic programs with sublinear space (or goal-size) complexity, for a special class of logic programs, which they call "logic programs with input". However, their result is not a natural improvement of Shapiro's since their logic programs have a strongly restricted form. Ruzzo (1980, 1981) showed that indexing alternating Turing machines (indexing ATMs), which are a "random access input" variation of ATMs, have a close relationship with other parallel complexity classes, especially those of uniform combinational circuits like *NC*, while relationships between logic programs and indexing ATMs have never been discussed.

In this paper, we improve Shapiro's (and also Stepanek-Stepankova's) results on relations between logic programs and alternating Turing machines, and characterize well-known parallel complexity classes in terms of logic programs. The main difference of our formulation from Shapiro's is that goal-size of logic programs is defined with the use of pointers taken into account. In our definition, the size of a term which occurs as a subterm in the initial goal clause can be estimated as the bit-length of the pointer representing it. Hence sublinear goal-size complexity is introduced naturally, just like space complexity of off-line Turing machines. This also make the random accessibility of indexing ATMs essential.

Two main theorem are derived using our new definition of goal-size complexity. One is an extension of Shapiro's first result to the case of sublinear space; logic programs of goal-size $G(n)$ and depth

machines with space $G(n)$ and in time $D(n)G(n)$. This is achieved by extending Shapiro's simulation to the case of indexing ATMs. The other is an improvement of Shapiro's second result; indexing ATMs using space $S(n)$ and tree-size $Z(n)$ are simulated by logic programs with goal-size $S(n)$ and depth $\log Z(n)$, which is achieved by utilizing Ruzzo's techniques for simulating ATMs (Ruzzo 1980).

These two results imply a close relationship between logic programs and tree-size bounded alternating Turing machines. It can be said that simultaneous goal-size/depth of logic programs is the same as space/tree-size of indexing ATMs, with goal-size and space up to a constant factor and likewise depth and log(tree-size). Some well-known complexity classes such as *LOGCFL*, *NC*, *PTIME* and *NPTIME* can be characterized via logic programs. In particular, it is shown that context-free languages can be recognized by logic programs with depth $O(\log n)$ and goal-size $O(\log n)$ simultaneously.

In the next section, we will give several basic definitions on logic programs and alternating Turing machines (Yasuura 1984). In Section 3, we will describe complexity measures of logic programs introduced by Shapiro, and then modify them. Simulations between logic programs and indexing Turing machines will be presented in Section 4. In Section 5, classes of languages recognized by logic programs will be considered.

2 BASIC CONCEPTS

2.1 Logic Programs

Let F be a finite set of *function symbols*, and V be a set of *variables*. We assume that $F \cap V = \emptyset$. Each function symbol is characterized by its name and its arity. Zero-arity function symbols are called *constants*. Variables are distinguished by an initial capital letter.

Terms on $F \cup V$ are defined recursively as follows:

(1) A variable $X \in V$ or a constant $a \in F$ is a term,

(2) If $t_1,...,t_k$ are terms and $f \in F$ is a k-arity function symbol, $f(t_1,...,t_k)$ is a term.

(3) All terms are generated by applying the above rules (1) and (2).

A term $p(t_1,...,t_k)$ is called an *atomic formula* if $p \in F$ is an k-arity predicate symbol. An atomic formula is considered to be a term which has logical value. (Note that we regard a predicate symbol as a kind of function symbol.)

Let T be the set of terms on $F \cup V$. A *substitution* $\theta: V \rightarrow T$ is represented by a finite set of ordered pairs of terms and variables

$$\{(t_i,X_i) \mid t_i \text{ is a term, } X_i \text{ is a variable and no two pairs have the same variable as the second element}\}.$$

Applying a substitution θ to a term t, we represent the resulting term by $t\theta$, which is called an *instance* of t. A substitution θ is called a *unifier* for two terms t_1 and t_2, if and only if $t_1\theta = t_2\theta$. A unifier θ is said to be the *most general unifier* of t_1 and t_2 if and only if, for any unifier θ_1 of t_1 and t_2, $t_1\theta_1$ is an instance of $t_1\theta$ and $t_2\theta_1$ is an instance of $t_2\theta$. If two terms are unifiable then they have a unique most general unifier.

Let $A, B_1, ..., B_k$ $(k \geq 0)$ be atomic formulae. A formula

$$A \leftarrow B_1,...,B_k$$

is called a *Horn clause* whose left side means the conjunction of B_i's. Particularly, we write

$$A \leftarrow$$

when $k=0$. A *logic program* is a finite set of Horn clauses. A conjunction of atomic formulae "$A_1,...,A_m$", $m \geq 0$, is called a *goal clause*, or simply a *goal*. When $m=1$, "A_1" is called a *unit goal*. When $m=0$, we denote it "$\Box$" and call it an *empty goal*.

We define computations of logic programs. Let N= "$A_1,A_2,...,A_m$", $m \geq 0$, be a (conjunctive) goal and C= "$A \leftarrow B_1,...,B_k$", $k \geq 0$, be a Horn clause such that A and A_i are unifiable via a substitution θ, for some $1 \leq i \leq m$. Then

$$N' = (A_1,...,A_{i-1},B_1,...,B_k,A_{i+1},...,A_m)\theta$$

is said to be *derived* from N and C with substitution θ.

Let P be a logic program and N be a goal. A *derivation* of N from P is a (possibly infinite) sequence of triples $<N_i,C_i,\theta_i>$, $i=0,1,...$, such that N_i is a goal, C_i is a clause in P, θ_i is a substitution, $N_0=N$ and N_{i+1} is derived from N_i and C_i with substitution θ_i, for all $i \geq 0$.

A derivation of N from P is called a *refutation* of N from P if $N_l = \Box$ (the empty goal) for some $l \geq 0$. Such a derivation is finite and of length l. If there is a refutation of a goal A from a program P, we also say that P *solves* A. Let R be a refutation of A from P. The *refutation tree* of R is a tree of unit goals, which is defined as follows:

(1) The root of the tree is A.

(2) Leaves are empty goals.

(3) In each step of derivation $<N_i,C_i,\theta_i>$, if C_i= "$A_i \leftarrow B_1,...,B_k$", and unit goal A_{ij} in N_i is unified with A_i by θ_i, then the node A_i has directed edges to all $B_j\theta_i$'s.

An example of logic programs, refutations and refutation trees are shown in Fig. 1.

The *Herbrand universe* of P, $H(P)$, is the set of all variable-free goals constructed by function symbols that occurs in P. The *Herbrand base* of P, $HB(P)$, is the set of all atomic formulae in $H(P)$. We define the *interpretation* of P, $I(P)$, to be the set $\{A \in HB(P) \mid P \text{ solves } A\}$.

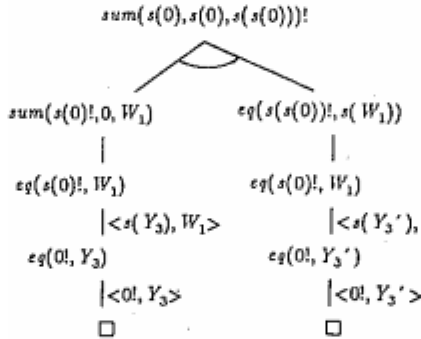
$eq(0,0) \leftarrow$

$eq(s(X),s(Y)) \leftarrow eq(X,Y)$

$sum(X,0,Y) \leftarrow eq(X,Y)$

$sum(X,s(Y),Z) \leftarrow sum(X,Y,W),\ eq(Z,s(W))$

(a) An example of logic programs



Goal: $"sum(s(0),s(0),s(s(0)))!"$

$< "sum(s(0),s(0),s(s(0)))!",$
$\quad "sum(X_1,s(Y_1),Z_1) \leftarrow sum(X_1,Y_1,W_1),\ eq(Z_1,s(W_1))",$
$\quad \{<s(0)!,X_1>,<s(0)!,Y_1>,<s(s(0))!,Z_1>\} >$

$< "sum(s(0)!,0,W_1),\ eq(s(s(0))!,s(W_1))",\ "sum(X_2,0,Y_2) \leftarrow eq(X_2,Y_2)",$
$\quad \{<s(0)!,X_2>,<W_1,Y_2>\} >$

$< "eq(s(0)!,W_1),\ eq(s(s(0))!,s(W_1))",\ "eq(s(X_3),s(Y_3)) \leftarrow eq(X_3,Y_3)",$
$\quad \{<0!,X_3>,<s(Y_3),W_1>\} >$

$< "eq(0!,Y_3),\ eq(s(s(0))!,s(W_1))",\ "eq(0,0) \leftarrow",\ \{<0!,Y_3>\} >$

$< "eq(s(s(0))!,s(s(0!)))",\ "eq(s(X_4),s(Y_4)) \leftarrow eq(X_4,Y_4)",\ \{<s(0)!,X_4>,<s(0!),Y_4>\} >$

$< "eq(s(0)!,s(0!)))",\ "eq(s(X_5),s(Y_5)) \leftarrow eq(X_5,Y_5)",\ \{<0!,X_5>,<0!,Y_5>\} >$

$< "eq(0!,0!)",\ "eq(0,0) \leftarrow",\ \emptyset >$

$< \Box, \Box, \emptyset >$

(b) An example of a refutation and a refutation tree

Figure 1. An example of logic programs and its computation
(The read-only annotation "!" will be defined in Section 3.)

2.2 Alternating Turing Machines

We assume familiarity with deterministic and non-deterministic Turing Machines (DTMs and NTMs, respectively). We will also use *alternating Turing machines* (ATMs) (Chandra et al. 1981) as our computation model.

ATMs are a generalization of nondeterministic Turing machines described informally as follows. The states of an ATM are partitioned into "existential" and "universal" states. As with an NTM, we can view a computation of an ATM as a tree of configuration. A *full computation tree* of an ATM M on a string w is a (possibly infinite) tree whose nodes are labeled with configurations of M on w, such that the descendants of any non-leaf node includes all of the successors of that configuration. A *computation tree* of M is a subtree of the full computation tree such that the descendants of any non-leaf node labeled by universal configurations includes all of the successors of that configuration, and the descendants of any non-leaf node labeled by existential configurations includes one of the successors of that configuration. An *accepting computation tree* is a finite computation tree whose leaf nodes are accepting configurations. M accepts w if and only if there exists an accepting computation tree whose root node is labeled with the initial configuration of M on w. Formal definitions of ATMs are found in (Chandra et al. 1981).

On-line ATMs (which have a writable input tape) and off-line ATMs (which have a read-only input tape and some work tapes) are defined similarly to the case of DTMs or NTMs. We will also use a "random access input" variation of ATMs called *indexing ATMs*, introduced by Ruzzo (1980). An indexing ATM has no input head; instead it has a special "index" tape and special "read" states. Whenever it enters a read state with an integer i written on the index tape, it reads the i-th symbol of the input and transit to an accepting or rejecting state. Thus in $O(\log n)$ steps, it can read any position of the input. There is only a constant loss in space and time in conversion of an on-line or off-line ATM to this normal form if space is at least $O(\log n)$.

An ATM uses *time* $T(n)$ (*tree-size* $Z(n)$) if for all accepted inputs of length n there is an accepting computation tree of height $\leq T(n)$ (respectively, size $\leq Z(n)$). An ATM uses *space* $S(n)$ if for all accepted inputs there is an accepting computation tree, each of whose nodes is labeled by a configuration using space $\leq S(n)$, and uses *alternation depth* $A(n)$ if for all accepted inputs there is an accepting computation tree, whose alternations of existential and universal configurations $\leq A(n)$.

We will denote the class of languages recognized by indexing ATMs within space $O(S(n))$ and time $O(T(n))$ simultaneously, by $A\!-\!SpTi(S(n),T(n))$. Likewise, $A\!-\!SpSz(S(n),Z(n))$ denotes languages recognized by ATMs running in space $O(S(n))$ and tree-size $O(Z(n))$ simultaneously, and similarly $A\!-\!SpAl(S(n),A(n))$ etc.

3 COMPLEXITY MEASURES FOR LOGIC PROGRAMS

First we describe complexity measures of logic programs introduced by Shapiro (1984). Let P be a logic program and A_0 be a unit goal. Here we regard P as a device which determines whether P solves a given goal or not. We consider that a goal A_0 is a input string for P, and P accepts A_0 if and only if P solves A_0. A goal is to a program what a input tape is to a Turing machine. The interpretation $I(P)$ is considered to be the "language" recognized by the program P. (P itself corresponds to, as it were, a finite control of a Turing machine.)

Assume that P solves A_0 with a refutation R. We regard R as a computation of P for input A_0. The *length* of R is defined as the number of nodes in the refutation tree of R, and the *depth* of R is defined as the height of the refutation tree. The *goal-size* of R is defined as the maximum size of any node in the refutation tree, where the size of goal is the number of symbols in its prefix notation. These three are considered to be "parallel" computation time, "serial" computation time and space required in the computation R, respectively. We say that a logic program P is of *goal-size complexity* $G(n)$, if any goal A_0 in $I(P)$ of size n has a refutation from P of goal-size $\leq G(n)$. *Depth complexity* and *length complexity* of P are defined similarly.

Shapiro showed the following relations between on-line alternating Turing machines and logic programs.

PROPOSITION 1. (Shapiro 1984) Let P be a logic program of depth complexity $D(n)$, goal-size complexity $G(n)$ and length complexity $L(n)$. Then there exists an on-line ATM M recognizing $I(P)$ such that M operates in time $O(D(n)G(n))$, space $O(G(n))$ and tree-size $O(L(n)G(n))$.

PROPOSITION 2. (Shapiro 1984) Let M be an on-line ATM that recognizes a language L in time $T(n)$, space $S(n)$ and tree-size $Z(n)$. Then there exists a logic program P of depth complexity $O(T(n))$, goal-size complexity $O(S(n))$ and length complexity $O(Z(n))$ such that $L=\{w\in\Sigma^* \mid accept(w)$ is in $I(P)\}$.

Note that any logic program satisfies $G(n)=\Omega(n)$ and any on-line ATM satisfies $S(n)=\Omega(n)$ and $T(n)=\Omega(n)$ from the definitions.

Goal-size is linearly related to alternating space. Depth and length are related to alternating time and alternating tree-size respectively, up to the goal-size factor. However, in Shapiro's definition of goal-size, "input" and "work space" required in the computation are not distinguished, and therefore goal-size is at least n, where n is the input length. We will modify the definition of goal-size to consider sublinear goal-size complexity.

Let D be a derivation of a unit goal A_0 from a program P. *Read-only* terms in D are variable-free terms occurring only in goal clauses and satisfying the following properties:

(1) Initial goal A_0 is a read-only term.

(2) Let A be a unit goal and A_i be the left-side of a Horn clause in P such that A and A_i are unified via the most general unifier θ in a step of the derivation D. If $(t,X)\in\theta$ and a subterm s of the term t occurs also in A as a read-only term, then s is a read-only term.

(3) All read-only terms in D are generated by applying the above rules (1) and (2).

Terms which are not read-only are called *writable*. A read-only term which is not a subterm of any read-only term is called a *primary read-only* term. We will mark primary read-only terms with "!". Note that any read-only term occurring in the derivation D of A_0 also occurs in A_0. An example of read-only terms are shown in Figure 1 in Section 2.1.

We now define the off-line goal-size of the refutation R of an primary read-only goal A_0. The size of a primary read-only term is defined as the smaller of the number of function symbols occur in it and $\lceil \log_2 n \rceil$, where n is the number of symbols in the initial goal A_0. The "off-line size" of a goal in R is the sum of the number of writable subterms and the size of all primary read-only terms. The size of writable terms is the number of symbols in its textual representation, while the size of read-only terms may be estimated as the number of symbols required to represent the address of the occurrence in binary representation if it is smaller. The *off-line goal-size* of a refutation is defined as the maximum off-line size of any node in the refutation tree of R.

According to this definition, the *off-line goal-size complexity* is defined similarly to the original goal-size complexity. Obviously our definition of the off-line goal-size complexity is a natural extension of Shapiro's definition. Henceforth, we say simply goal-size complexity as off-line goal-size complexity.

4 ATMS AND LOGIC PROGRAMS

4.1 Simulating Logic Programs with ATMs.

We will describe an algorithm for simulating a logic program by an indexing alternating Turing machine, which is a modification of Shapiro's method (Shapiro 1984), and show that sublinear goal-size logic programs can be simulated by an indexing ATM in sublinear space.

Without loss of generality, we assume that arity of any function symbol is at most 2. Consider Algorithm SIMULATE1 shown in Fig. 2, which answers whether a unit goal A is proved by a program P. DERIVE is a procedure for the derivation and UNIFY is a subroutine for unification. In our simulation, goals are stored in a work tape of an indexing ATM,

```
Algorithm SIMULATE1
Given: a logic proaram P
Input: a unit goal A
Output: whether P proves A or not

function UNIFY( t: a variable free term, t' : a term ) : a substitution ;
 begin
   case( t' ) of
(U1)    a variable "X" : UNIFY:= {(X,t')}
(U2)    a constant "a" : if t="a" then UNIFY:= ϕ else reject;
       an 1-arity function symbol "f(t1')" :
(U3)          if ( t="f(t1)" ) then UNIFY:=UNIFY(t1,t1') else reject;
       a 2-arity function symbol "g(t1',t2')" :
            if ( t="g(t1,t2)" ) then
               begin
(U4)                Guess the location of t2 (existential branch);
(U5)                if the guessd term is incorrect then reject (universal branch);
(U6)                UNIFY1:=UNIFY(t1,t1'); UNIFY2:=UNIFY(t2,t2');
(U7)                if ( UNIFY1 and UNIFY2 are inconsistent ) then reject;
(U8)                UNIFY:=UNIFY1⋃UNIFY2
               end
            else reject;
   end {of case}
 end {of function UNIFY};

procedure DERIVE ( A : a unit goal );
 begin
 (R1) Choose a clause C="A' ←B1,...,Bk'' (k≥0) in P (existential branch);
 (R2) σ:=UNIFY(A,A');
 (R3) Guess a substitution θ (existential branch)
        such that all remaining variables in Cσ occur as the second elements
        and that no variable occurs in the first elements ;
 (R4) for all i∈{1,...,k} parallel do (universal branch)
        DERIVE ( Biσθ );
 end { of function DERIVE };

begin {of main routine}
   Output "YES" if DERIVE( A ) is end
end;
```

Figure 2. An algorithm for simulating logic programs by indexing ATMs.

where writable terms are represented in prefix notation and primary read-only terms may be represented as a pointer to the input string.

UNIFY is a function which returns the most general unifier of a variable-free term t and a term t'. First we consider the case that t is writable. If t' is a variable X, it returns the substitution $\{(X,t)\}$ (in line (U1)). If t' is a constant a and $t=a$, it returns ϕ (in line (U2)). If t' is $f(t_1')$ and t is $f(t_1)$, where f is a 1-arity function symbol and t_1' and t_1 are terms, it returns the result of UNIFY(t_1,t_1') recursively (in line (U3)). If t' is $g(t_1',t_2')$ and t is $g(t_1,t_2)$, where g is a 2-arity function symbol and t_1', t_2', t_1, t_2 are terms, it scans the term $g(t_1,t_2)$ represented in prefix notation on the work tape, guesses the second argument t_2 nondeterministically (in line (U4)), and verifies it using universal branches (in line (U5)). Next it calls the subroutines UNIFY(t_1,t_1') and UNIFY(t_2,t_2') (in line (U6)), verifies if no variable is substituted by different terms (in line (U7)) and returns the union of them (in line (U8)). In every other case it answers that these two terms are not unifiable. It is almost similar in the case that t' is read-only term represented as a pointer, except that arguments are passed by their addresses; however, in case that $t'=g(t_1',t_2')$ and $t=g(t_1,t_2)$, the address of t_2 on the input tape is guessed nondeterministically (in line (U4)), instead of scanning the work tape.

Procedure DERIVE chooses a Horn clause $C=``A' \leftarrow B_1,...,B_k"$ $(k \geq 0)$ in P nondeterministically (in line (R1)) and get the most general unifier σ by calling Function UNIFY (in line (R2)). Suppose A and A' are unifiable. Since no variable occurs in goal A, all variables occur in A' are substituted by variable-free terms via σ. Next it guesses a variable-free substitution θ of variables which are not substituted via σ (in line (R3)). All variables occur in C are substituted by variable-free terms via $\sigma\theta$. Finally, it asks if $B_i\sigma\theta$ can be derived by P for each B_i using universal branches (in line (R4)).

Let us consider the logic program P_{real} shown in Fig. 4. Obviously P_{real} behaves quite similar to Algorithm SIMULATE2. From Proposition 3, if M accepts w within space $\leq S(n)$ and tree-size $\leq Z(n)$, then there exists a refutation of

$$realizable(\ q_0(\ w!,\ \$,\ \$)\ )$$

from P_{real} with depth $O(\log Z(n))$, goal-size $O(Z(n))$ and length $O(Z(n))$. Thus the following theorem follows:

THEOREM 2. Let M be an ATM which accepts a language L in space $S(n) \geq \Omega(\log n)$ and tree-size $Z(n)$. Then there exists a logic program P of depth complexity $O(\log Z(n))$, goal-size complexity $O(S(n))$ and length complexity $O(Z(n))$ such that $\{w \in \Sigma^* | accept(w)$ is in $I(P)\}$, is a language recognized by M.

Alternating space is linearly related to the goal-size. Alternating tree-size is linearly related to the length and logarithm of alternating tree-size is linearly related to the depth. The next corollary follows immediately from the theorem.

COROLLARY 1. Let M be an ATM that accepts a language L in time $T(n)$, space $S(n) \geq \Omega(\log n)$ and tree-size $Z(n)$. Then there exists a logic program P of depth complexity $O(T(n))$, goal-size complexity $O(S(n))$ and length complexity $O(Z(n))$ such that $\{w \in \Sigma^* | accept(w)$ is in $I(P)\}$ is a language recognized by M.

Remark. Note that $T(n) \geq O(\log Z(n))$. □

Since Corollary 1 subsumes Proposition 2, Theorem 2 is an improvement of Proposition 2.

4.2 Simulating ATMs with Logic Programs

We will describe an algorithm for simulating indexing alternating Turing machines by logic programs. This algorithm is based on a simulation technique introduced by Ruzzo (1980) for simulating an $S(n)$-space and $Z(n)$-tree-size bounded ATM (which may be indexing, off-line or on-line) by an $O(S(n)\log Z(n))$-time bounded and $O(S(n))$-space bounded indexing ATM.

A computation *fragment* of an $S(n)$-space and $Z(n)$-tree-size bounded ATM M is an ordered pair (r,L), where r is a configuration of M using space $\leq S(n)$, and L is a set of such configurations. A fragment is *realizable* if there is a computation tree of M with root r whose leaves are either accepting or in L. An algorithm for deciding realizability of a fragment (r,L). Algorithm SIMULATE2, is shown in Fig. 3.

PROPOSITION 3. (Ruzzo 1980). If ATM M accepts w within tree-size $\leq Z(n)$, REAL$(r_0,\emptyset)$ returns "true" with maximum depth of recursions $O(\log Z(n))$.

Ruzzo also showed that it is sufficient to consider fragments whose corresponding computation trees have not more than three non-accepting leaves in Algorithm SIMULATE2.

Next we will describe a logic program which simulates Algorithm SIMULATE2. We may assume that M is an indexing ATM which has one work tape and that the index tape alphabet is $\{0,1\}$. In our simulation, an input string $w=x_1x_2...x_n \in \Sigma^*$ of length n is represented by a term

$$\bullet(...(\bullet(\bullet(x_1,x_2),\bullet(x_3,x_4)),...,\bullet(\bullet(x_n,\$),\bullet(\$,\$))...)$$

in complete-binary-tree form of height $\lceil \log_2 n \rceil$, where "$\bullet$" is a 2-arity function symbol and "\$" is a con-

```
Algorithm SIMULATE2
Given: an indexing alternating Turing machine M
Input: a string w∈Σ*
Output: whether M accepts w or not

function REAL( R: an ATM configuration, L: a set of ATM configuration ): Boolean;
   begin
(F1) if the fragment (r,L) is realizable within tree-size ≤3 then
          REAL := true
      else
         begin
(F2)       Guess an ATM configuration s and sets of ATM configurations L', L"
              s.t. L'⊆L, L"⊆L and L=L'∪L" (existential branch) ;
(F3)       REAL := ( REAL( r, L'∪{s} ) ∧ REAL( s, L" ) ) (universal branch)
         end
   end {of REAL};

begin
   Output "YES" if REAL( r0, φ ) = TRUE,
      where r0 is an initial configuration of M with input w;
end.
```

Fig. 3. An algorithm for Simulating ATMs

stant. A configuration of indexing ATM M is represented by a term

$$q(\ w_{cur}!,\ t_{left},\ t_{right}\ )$$

where q corresponds the state, w_{cur} is a subtree of w obtained by traversing a complete binary tree w according to the content of the index tape, and t_{left} and t_{right} represent the left side (including the head position) and the right side of the content of the work tape respectively.

Let n be the length of A_0, and let $G(n)$, $D(n)$ and $L(n)$ be the goal-size, depth and length of R respectively. By examining how the refutation R of A_0 from P is simulated by an indexing ATM, it is not difficult to show that the total space, time, tree-size and alternation depth required in the simulation are $O(G(n)+\log n)$, $O(D(n)G(n)+(\log n)^2)$, $O(L(n)\ (G(n)+n(\log n)^2))$ and $O(D(n)+\log n)$ respectively.

THEOREM 1. A logic program of goal-size complexity $G(n)<O(n)$, depth complexity $D(n)\geq\Omega(\log n)$, length complexity $L(n)$, can be simulated by an indexing ATM in time $O(D(n)G(n))$, space $O(G(n))$, tree-size $O(L(n)\cdot n(\log n)^2)$ and alternation depth $O(D(n))$.

Goal-size is linearly related to the alternating space, depth is related to the alternating time up to the goal-size factor (indeed it is linearly related to the alternation depth) and length is related to the alternating tree-size up to the polynomial factor. This is very similar to Proposition 1, and we may say that Theorem 1 extends Proposition 1 to the case that goal-size complexity $G(n)=o(n)$.

Program P_{real}

$$accept(W)\leftarrow realizable(q_0(W,\$,\$),set(\$,\$,\$))$$

$$realizable(Y,set(Y1,Y2,Y3))\leftarrow realizable(Y,set(Z,Y2,Y3)),\ realizable(Z,set(Y1,Y2,Y3))$$

$$realizable(Y,set(Y1,Y2,Y3))\leftarrow realizable(Y,set(Y2,Y1,Y3))$$
$$realizable(Y,set(Y1,Y2,Y3))\leftarrow realizable(Y,set(Y3,Y2,Y1))$$

$$realizable(Y,set(Y1,Y2,Y3))\leftarrow realizable(Y,set(Y,Y2,Y3))$$

and clauses which correspond to the state transitions each of the ATM M.

Figure 4. Program for simulating ATMs

5 LANGUAGES RECOGNIZED BY LOGIC PROGRAMS

We will examine the relations among classes of languages recognized by logic programs and many well-known complexity classes.

We denote the class of languages recognized by logic programs within goal-size $O(G(n))$, depth $O(D(n))$ and length $O(L(n))$ simultaneously, by $L\text{-}GzDpLn(G(n),D(n),L(n))$. Likewise, $L\text{-}GzDp(G(n),D(n))$ denotes languages recognized by logic programs within goal-size $O(G(n))$ and depth $O(D(n))$ simultaneously, and similarly for $L\text{-}GzLn(G(n),L(n))$ etc.

THEOREM 3. For $D(n)\geq\Omega(\log n)$ and $G(n)\leq 2^{O(D(n))}$,

$$L\text{-}GzDp(G(n),D(n))=L\text{-}GzLn(G(n),2^{O(D(n))})$$
$$=A\text{-}SpSz(G(n),2^{O(D(n))}).$$

Proof. From Proposition 1 and Theorem 1,

$$L\text{-}GzLn(G(n),2^{O(D(n))})$$
$$\subseteq A\text{-}SpSz(G(n),\ 2^{O(D(n))}\cdot(G(n)+n\log^2 n))$$
$$=A\text{-}SpSz(G(n),2^{O(D(n))}).$$

From Theorem 2,

$$A\text{-}SpSz(G(n),2^{O(D(n))})$$
$$\subseteq L\text{-}GzDpLn(G(n),D(n),2^{O(D(n))})$$

It is obvious that, if the depth of the refutation is $D(n)$, the length of a refutation is at most $2^{O(D(n))}$, i.e.

$$L\text{-}GzDp(G(n),D(n))$$
$$\subseteq L\text{-}GzDpLn(G(n),D(n),2^{O(D(n))}).$$

□

COROLLARY 2. The class of languages recognized by logic programs within goal-size $O(\log n)$ and depth $O(\log n)$ is $LOGCFL$, i.e. the class of languages log-space reducible to context free languages.

$$L\text{-}GzDp(\log n,\log n)=L\text{-}GzLn(\log n,2^{O(\log n)})$$
$$=A\text{-}SpSz(\log n,2^{O(\log n)})$$
$$=LOGCFL.$$

Remark. The corollary immediately follows from Theorem 3 for $G(n)=O(\log n)$ and $D(n)=O(\log n)$. □

Similarly, the following result follows immediately.

$$L\text{-}GzDp(\log n,(\log n)^{O(1)})=L\text{-}GzLn(\log n,2^{(\log n)^{O(1)}})$$
$$=A\text{-}SpSz(\log n,2^{(\log n)^{O(1)}})$$
$$=NC.$$

$$L\text{-}GzDp(\log n,n^{O(1)})=L\text{-}GzLn(\log n,2^{n^{O(1)}})$$
$$=A\text{-}SpSz(\log n,2^{n^{O(1)}})$$
$$=PTIME.$$

$$L\text{-}GzDp(n^{O(1)},\log n)=L\text{-}GzLn(n^{O(1)},n^{O(1)})$$
$$=A\text{-}SpSz(n^{O(1)},n^{O(1)})$$
$$=NPTIME.$$

Hierarchy of complexity classes are shown in Fig. 5. Here *NC* is the class of all problems solvable by uniform circuit families with depth $O((\log n)^{O(1)})$ and size $n^{O(1)}$; *LOGSPACE* (*NLOGSPACE*) is the class solvable by a DTM (respectively, an NTM) in space $O(\log n)$; *PTIME* (*NPTIME*) is the class solvable by a DTM (respectively, an NTM) in time $n^{O(1)}$; *PSPACE* is the class solvable by a DTM in space $n^{O(1)}$. (See (Cook 1985).)

6 CONSIDERATIONS

We have modified the definition of complexity of logic programs introduced by Shapiro, and have shown a relationship between the complexity of logic programs and that of alternating Turing machines. In particular, logic programs are closely related to tree-size bounded alternating Turing machines. We can say that logic programs are a surprisingly good model of parallel computation. Indeed, logics program can be considered as alternating Turing machines whose existential branches and universal branches are not symmetric.

NC is a very attractive class, because all problems in it are solvable very rapidly by circuits with polynomial gates. We have given a characterization of *NC* in terms of logic programs. Another characterization of *NC* via logic programs is found in (Ullman and Gelder 1986).

Theorem 3 means that any logic programs of length complexity $L(n)$ (whose depth complexity is $O(L(n))$) can be simulated by a logic program with depth complexity $\log(L(n))$ and length complexity $L(n)^{O(1)}$. This may be regard as conversion from "serial" logic programs into parallel form.

ACKNOWLEDGMENTS

We would like to express sincere appreciation to Dr. H. Hiraishi, Dr. N. Takagi, N. Ishiura, and all the members of the Yajima Laboratory at Kyoto University, for their helpful discussions. We would also like to thank Dr. H. Yasuura in Kyoto University for his comments and criticism.

REFERENCES

E. Y. Shapiro: "A Subset of Concurrent Prolog and its Interpreter", *ICOT Tech. Report TR-003*, (1983).

K. Ueda: "Guarded Horn Clauses, *Proc. of the Logic Programming Conference*, (1985).

E. Y. Shapiro: "Alternation and the Computational Complexity of Logic Programs", *J. Logic Programming*, **1**, 19-33, (1984).

A. K. Chandra, D. C. Kozen and L. J. Stockmeyer: "Alternation", *J. ACM*, **28-1**, (1981).

P. Stepanek and O. Stepankova: "Logic Programs and Alternation", *3rd International Conference on Logic Programming*, Lecture Notes in Computer Science, **225**, (1986).

W. L. Ruzzo: "Tree-Size Bounded Alternation", *Journal of Computer and System Sciences*, **21-2**, 218-235, (Oct. 1980)

W. L. Ruzzo: "On Uniform Circuit Complexity", *Journal of Computer and System Sciences*, **22-3**, 365-383, (June 1981).

H. Yasuura: "Parallel Computational Complexity of Logic Programs", *Report of Tech. Group on Automatons and Languages, IECEJ*, **AL83-69**, (Feb. 1984), in Japanese.

S. A. Cook: "A Taxonomy of Problems with Fast Parallel Algorithms", *Inform. and Control*, **64**, 2-22, (1985).

J. D. Ullman and A. van Gelder: "Parallel Complexity of Logical Query Programs", *27th IEEE FOCS*, 438-454, (1986).

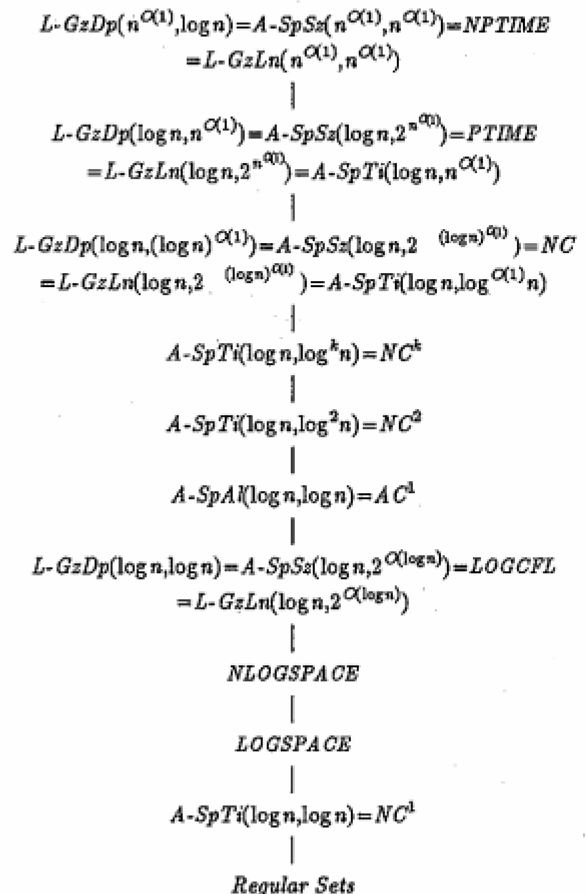


Figure 5. Hierarchy of complexity classes of problems with respect to logic program complexity