Alternation and the Computational Complexity of Logic Programs

June, 1982

Research Report 239

Ehud Y. Shapiro

Alternation and the Computational Complexity of Logic Programs

Ehud Y. Shapiro[1]

Department of Computer Science
Yale University
New Haven, CT 06520

Abstract.

We investigate the complexity of derivations from logic programs, and find it closely related to the complexity of computations of alternating Turing machines. In particular, we define three complexity measures over logic programs — goal-size, length and depth — and show that goal-size is linearly related to alternating space, the product of length and goal-size is linearly related to alternating tree-size, and the product of depth and goal-size is linearly related to alternating time. The bounds obtained are simultaneous.
As an application, we obtain a syntactic characterization of Nondeterministic Linear Space and Alternating Linear Space via logic programs.

1. Introduction

Since the introduction of the resolution principle by Robinson [18] there have been attempts to use it as the basic computation step in a logic-based programming language [4, 11]. Nevertheless, for general first order theories, neither resolution nor its successive improvements were efficient enough to make the approach practical. A breakthrough occured when a restricted form of logical theories was considered, namely Horn theories. Since the pioneering works of Colmerauer, van Emden and Kowalski [6, 9, 14], the idea of a procedural interpretation to Horn-clause logic has materialized. There is a growing body of theory of logic programming (e.g. [1, 9, 15, 16]), and the programming language Prolog [2, 19], which is based on this idea, is a viable alternative to the programming language Lisp in the domain of symbolic programming [17, 21].

[1]The author acknowledges the support of the National Science Foundation, grant No. MCS8002447.

Author's current address: Department of Applied Mathematics, Weizmann Institute of Science, Rehovot 76000, ISRAEL.

The model-theoretic, fixpoint and operational semantics of logic programs have been studied by Apt, van Emden and Kowalski [1, 9], among others. The current paper studies the computational complexity of logic programs. The results reveal similarities between logic programs and alternating Turing machines. Since the complexity of alternating Turing machines is well understood, these results provide a basis for evaluating the complexity of logic programs.

The application of these results provide a link between the structural complexity and computational complexity of logic programs, a relation rarely found among practical programming languages. The close relationship of logic programs to alternating Turing machines may also be considered as further evidence for their potential as a programming language for parallel machines [5, 7, 8].

Our goal in this work is to provide a theoretical basis for analyzing the computational complexity of concrete logic programs. The applications of our results suggest, however, that complexity theory in general may benefit from the study of this computational model.

2. Logic programs

2.1. Definitions and examples

A logic program is a finite set of *definite clauses*, which are universally quantified logical sentences of the form

$$A \leftarrow B_1, B_2, \ldots, B_k \qquad k \geq 0$$

where the A and the B's are logical atoms, also called *unit goals*. Such a sentence is read "A is implied by the conjunction of the B's", and is interpreted procedurally "to satisfy goal A, satisfy goals B_1 and B_2 and ... and B_k". A is called the clause's *head* and the B's the clause's *body*. If the B's are missing, the sentence reads "A is true" or "goal A is satisfied". Given a unit goal, or a conjunction of goals, a set of definite clauses can be executed as a program, using this procedural interpretation. Figure 1 establishes some relationships between logic programs and concepts from conventional programming languages.

An example of a logic program for quicksort is shown as Program 1. We use upper-case strings as variable symbols and lower-case strings for all other symbols. The term [] denotes the empty list, and the term $[X|Y]$ stands for a list whose head (car) is X and tail (cdr) is Y. The results of unifying the term $[A,B|X]$ with the list [1,2,3,4] is $A=1$, $B=2$, $X=[3,4]$, and unifying $[X|Y]$ with $[a]$ results in $X=a$, $Y=[]$.

In establishing the relationship between computations of alternating Turing machines and logic programs, we develop below logic programs that simulate alternating Turing machines. A

Procedures	Definite clauses
Procedure calls	Goals
Binding mechanism, data selectors and constructors	Unification
Execution mechanism	Nondeterministic goal reduction

Figure 1: Common programming concepts in logic programs

Program 1: Quicksort

```
qsort([X|Xs],Ys) ←
        partition(Xs,X,Ys1,Ys2), qsort(Ys1,Zs1), qsort(Ys2,Zs2), append(Zs1,[X|Zs2],Ys).
qsort([],[]).

partition([Z|Xs],X,Ys,[Z|Zs]) ← X<Z, partition(Xs,X,Ys,Zs).
partition([Y|Xs],X,[Y|Ys],Zs) ← X≥Y, partition(Xs,X,Ys,Zs).
partition([],X,[],[]).

append([X|Xs],Ys,[X|Zs]) ← append(Xs,Ys,Zs).
append([],Xs,Xs).
```

simpler precursor of these programs is Program 2. It is a logic program that simulates a two-state pushdown automaton that accepts palindromes over an arbitrary alphabet. The procedure *pal*(Q,X,Y) stores in Q the state of the automaton, in X the remaining input string and in Y the pushdown stack. The semantics of *pal*(Q,X,Y) is "The pda accepts the string X starting from state Q and stack contents Y." The program is designed to succeed on the goal *pal*($q0$,S,[]) iff S is a palindrome. The clauses in the program are of the form

accept(q,[A|X],S) ← *accept*(q',X,S').

Such a clause reads "The pda accepts [A|X] in state q and stack contents S if it accepts the string X in state q' and stack contents S'". The last clause, *pal*($q1$,[],[]), says "The pda accepts the empty string in state $q1$ and empty stack".

Program 2: Simulating a pushdown automaton

```
pal(q0,[A|X],Y) ← pal(q0,X,[A|Y]).
pal(q0,[A|X],Y) ← pal(q1,X,Y).
pal(q0,X,Y) ← pal(q1,X,Y).
pal(q1,[A|X],[A|Y]) ← pal(q1,X,Y).
pal(q1,[],[]).
```

2.2. Computations

A computation of a logic program P can be described informally as follows. The computation starts from some initial (possibly conjunctive) goal A; it can have two results: success or failure. If a computation succeeds, then final instantiations of the variables in A are conceived of as the output of the computation. A given goal can have several successful computations, each resulting in a different output.

The computation progresses via nondeterministic goal reduction. At each step we have some current goal $A_1,A_2,...,A_n$. A goal A_i and a clause $A' \leftarrow B_1,B_2,...,B_k$ in P are then chosen nondeterministically; the head of the clause A' is unified with A_i via a substitution θ, and the reduced goal is $(A_1,...,A_{i-1},B_1,B_2,...,B_k,A_{i+1},...,A_n)\theta$. The computation terminates when the current goal is empty.

We proceed to formalize these notions. We follow the Prolog-10 manual [2] in notational conventions, and Apt and van Emden [1] in most of the definitions. A *term* is either a constant, a variable, or a compound term. The constants include integers and atoms. The symbol for an atom can be any sequence of characters, which is quoted if there is possibility of confusion with other symbols (such as variables, integers). Variables are distinguished by an initial capital letter. If a variable is only referred to once, it does not need to be named and may be written as an "anonymous" variable indicated by a single underline _.

A *compound term* comprises a functor (called the principal functor of the term) and a sequence of one or more terms called arguments. A functor is characterized by its name, which is an atom, and its arity or number of arguments. An atom is considered to be a functor of arity 0.

A *substitution* is a finite set (possibly empty) of pairs of the form $X \rightarrow t$, where X is a variable and t is a term, and all the variables X are distinct. For any substitution $\theta = \{X_1 \rightarrow t_1, X_2 \rightarrow t_2, ..., X_n \rightarrow t_n\}$ and term s, the term $s\theta$ denotes the result of replacing each occurrence of the variable X_i by t_i, $1 \leq i \leq n$; the term $s\theta$ is called an *instance* of s.

A substitution θ is called a *unifier* for two terms s_1 and s_2 if $s_1\theta = s_2\theta$. Such a substitution is called the ***most general unifier*** of s_1 and s_2 if for any other unifier θ_1 of s_1 and s_2, $s_1\theta_1$ is an instance of $s_1\theta$. If two terms are unifiable then they have a unique most general unifier [18].

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 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. A goal $A_j\theta$ of N' is said to be ***derived*** from A_j in N. A goal $B_j\theta$ of N' is said to be ***invoked*** by A_i and C.

Let P be a logic program and N 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 with new variable symbols not occuring previously in the derivation, 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 $N_l=\square$ (the empty goal) for some $l\geq 0$. Such a derivation is finite and of length l, and we assume by convention that in such a case $C_l=\square$ and $\theta_l=\{\}$. If there is a refutation of a goal A from a program P we also say that ***P solves A***.

Figure 2 shows a refutation of the goal *pal*(*q0*,[*a*,*b*,*a*],[]) from Program 2.

$<pal(q0,[a,b,a],[]),\ pal(q0,[A0|X0],Y0) \leftarrow pal(q0,X0,[A0|Y0]),\ \{A0\rightarrow a,X0\rightarrow [b,a],Y0\rightarrow []\}>$

$<pal(q0,[b,a],[a]),\ pal(q0,[A1|X1],Y1) \leftarrow pal(q1,X1,Y1),\ \{A1\rightarrow b,X1\rightarrow [a],Y1\rightarrow [a]\}>$

$<pal(q1,[a],[a]),\ pal(q1,[A2|X2],[A2|Y2]) \leftarrow pal(q1,X2,Y2),\ \{A2\rightarrow a,X2\rightarrow [],Y2\rightarrow []\}>$

$<pal(q1,[],[]),\ pal(q1,[],[]),\ \{\}>$

$<\square,\square,\{\}>$

Figure 2: An example of a refutation

A more intuitive, though less complete way to describe a successful compution of a logic program (i.e. a refutation) is via a ***refutation tree***. In a refutation tree nodes are goals that occur in the computation, with their variables instantiated to their final values, and arcs represent the relation of goal invocation. The refutation tree that corresponds to the refutation in Figure 2 is simply the list of goals in this refutation, which are the first elements of the triples, connected with arcs. The refutation tree in Figure 3 corresponds to the refutation of *qsort*([2,1,3],*L*) from Program 1. Depth of indentation reflects depth in the tree.

```
qsort([2,1,3],[1,2,3])
    partition([1,3],2,[1],[3])
        2≥1
        partition([3],2,[],[3])
            2<3
            partition([],2,[],[])
    qsort([1],[1])
        partition([],1,[],[])
        qsort([],[])
        qsort([],[])
        append([],[1],[1])
    qsort([3],[3])
        partition([],3,[],[])
        qsort([],[])
        qsort([],[])
        append([],[3],[3])
    append([1],[2,3],[1,2,3])
        append([],[2,3],[2,3])
```

Figure 3: An example of a refutation tree

2.3. Semantics

We define semantics of logic programs, which is a special case of the standard model-theoretic semantics of first order logic [9]. An ***interpretation*** is a set of variable-free goals. The ***Herbrand universe*** of ***P***, ***H(P)***, is the set of all variable-free goals constructable from constants and functors that occur in ***P***. We define the ***interpretation of P***, ***I(P)***, to be the set $\{A|\ A \in \boldsymbol{H(P)}$ and $\boldsymbol{P}$ solves $A\}$. Van Emden and Kowalski [9] show that ***I(P)*** is the minimal model in which ***P*** is true. They also associate a transformation τ_P with any program ***P***, and show that ***I(P)*** is the least fixpoint of τ_P The transformation τ_P is defined as follows. Let ***I*** be a subset of ***H(P)***. Then a variable-free goal $A \in \boldsymbol{H(P)}$ is in $\tau_P(I)$ iff there is a variable-free instance $A \leftarrow B_1, B_2, .. B_k$ of a clause in ***P*** such that B_i is in ***I*** for all i, $1 \leq i \leq k$.

$U \subseteq Q$ is the set of universal states, and
$Q - U$ is the set of existential states.

An alternating Turing machine is called a *nondeterministic Turing machine* if it has at most one transition for any universal state and k-tuple of tape symbols.

A *step* of M consists of reading one symbol from each tape, writing a symbol on each tape, and moving each of the heads left or right one tape cell, in accordance with the transition relation δ.

A *configuration* of an ATM M is an element of $Q \times (\Gamma^*)^{2k}$, representing the state of the finite control, the nonblank content of the k tapes to the left of the k heads, and the content of the k tapes to the right of the k heads, including the symbols on which the heads are positioned. A configuration is called *universal* if its state is in U, *existential* if its state is in $Q-U$.

A configuration β is a *successor* of a configuration α if β follows from α in one step, according to the transition rule δ. A *computation path* $\alpha_1, \alpha_2,\ldots$ is a (possibly infinite) sequence of configurations of M for which α_{i+1} is a successor of α_i, for all $i \geq 1$.

A *computation tree* of M is a rooted, directed tree whose nodes are configurations of M and every path in the tree is a computation path of M. A computation tree T of M is *complete* if it has the following properties:

1. For every universal configuration α in T and every successor β to α there is an edge (α,β) in T.
2. All the leaves of T are universal configurations.

A configuration α *leads to acceptance* if it is the root of a finite, complete computation tree.

A computation tree T *accepts* a string x if it is finite, complete, and its root is the configuration $<q_0,[]^k,x,[]^{k-1}>$, where $[]$ denotes the empty string.

We say that M *accepts* x if it has a computation tree that accepts x, and define $L(M)$ to be the set of strings accepted by M.

The space of a configuration is the sum of lengths of the nonblank tape contents of the configuration. The *space* of a computation tree T is the maximum space of any configuration in T. The *time* of T is the maximum length of any path in T. The *size* of T is the number of nodes in T.

An alternating Turing machine M *operates in space* $S(n)$ if for every string $x \in L(M)$ of length n there is a computation tree of M of space $S(n)$ that accepts x. Similarly, M *operates in time* $T(n)$ if for every string $x \in L(M)$ of length n there is a computation tree of M of time $T(n)$ that accepts x. M *operates in tree-size* $Z(n)$ if for every string $x \in L(M)$ of length n there is a computation tree of M of size $Z(n)$ that accepts x (cf. [20]). Note that we measure only accepting computations.

2.4. Complexity measures

We define complexity measures over refutations, using the notion of refutation tree. Let R be a refutation. We define the *length* of R to be the number of nodes in the refutation tree. The *depth* of R is the depth of the tree. The *goal-size* of R is the maximum size of any node of the refutation tree, where the size of a goal is the number of symbols in its textual representation.

Definition 2.1: We say that a logic program P is of *goal-size complexity* $G(n)$ if for any goal A in $I(P)$ of size n there is a refutation of A from P of goal-size $\leq G(n)$.

P is of *depth complexity* $D(n)$ if for any goal A in $I(P)$ of size n there is a refutation of A from P of depth $\leq D(n)$.

P is of *length complexity* $L(n)$ if for any goal A in $I(P)$ of size n there is a refutation of A from P of length $\leq L(n)$.

We say that an interpretation I is of goal-size complexity $G(n)$ if there is a logic program P such that $I(P){\models}I$ and the goal-size complexity of P is $G(n)$. We assume similar definitions for the depth complexity and length complexity of interpretations.

3. Alternating Turing machines

Alternating Turing machines, introduced by Chandra, Kozen, and Stockmeyer [3], generalize nondeterministic Turing machines. An alternating Turing machine (ATM) is a Turing machine with two types of states, existential and universal. An ATM in an existential state functions similarly to a nondeterministic Turing machine: it accepts if and only if at least one of its applicable next moves leads to acceptance; in particular, it rejects it has no applicable next move. An ATM in a universal state accepts if and only if each of its applicable next moves leads to acceptance; in particular, it accepts it has no applicable next move. When discussing computations informally, we adopt the procedural point of view that a process (=configuration) in an existential state spawns a new process for any of its applicable next moves, and accepts if at least one of them accepts, and that a process in a universal state spawns a new process for any of its applicable next moves, and accepts only if all of them accept.

For completeness, we provide a formal definition of an ATM, adapted from Chandra et al. [3], and Fischer and Ladner [10]. A *k-tape alternating Turing machine* is a seven-tuple $M=(k,Q,\Delta,\Gamma,\delta,q_0,U)$, where

Q is the set of states,
Δ is the input alphabet,
Γ is the tape alphabet,
$\# \in \Gamma - \Delta$ is the blank symbol,
$\delta \in (Q\times\Gamma^k)\times(Q\times\Gamma^k\times \{left, right\}^k)$ is the next move relation,
$q_0 \in Q$ is the initial state,

The fundamental results of Chandra et al. relate the computational complexity of alternating Turing machines to those of deterministic Turing machines.

4. Simulations among alternating Turing machines and logic programs

The similarity between an abstract interpreter of logic programs and the execution mechanism of alternating Turing machines is quite apparent. The existential state of the ATM corresponds to the nondeterministic choice of a clause whose head unifies with a goal. The universal state corresponds to the simultaneous satisfaction of the the goals in the body of the clause. A goal immediately fails if the head of no clause unifies with it; a Turing machine rejects if it is in an existential state with no applicable next move. A goal immediately succeeds if it is unifiable with a unit clause — a clause with an empty body; an ATM accepts if it is in a universal state with no applicable next move.

The remainder of this section provides these intuitions with a precise foundation. We describe simulations between logic programs and alternating Turing machines, and use them to relate the complexity measures defined over logic programs to complexity measures over alternating Turing machines.

4.1. Simulating a Logic Program with an Alternating Turing Machine

In Simulation 1 a logic program P is simulated by an ATM M. M uses existential branching to nondeterministically choose both the next clause to be invoked and the unifying substitution, and universal branching to simultaneously satisfy all the goals in the body of the clause.

Simulation 1: An alternating Turing machine simulates a logic program

Let P be a logic program. We describe an ATM M with the property that for any variable-free unit goal A, M accepts A iff P solves A.

The ATM M stores P in its finite control, and initially has A written on its tape. From its initial state it proceeds as follows: using existential branching, it chooses a clause $A' \leftarrow B_1, B_2, \ldots, B_k$, $k \geq 0$, in P, and writes on its tape a substitution θ. It then computes $A'\theta$, verifies that $A = A'\theta$, and erases everything from the tape except θ. Then, using universal branching, it chooses B_i for some i, $0 \leq i \leq k$, applies θ to B_i, erases everything from the tape except $B_i\theta$ and returns to its initial state. ▯

Note that M accepts if the invoked clause has an empty body, and rejects if it fails to find a clause in P whose head unifies with its current goal. Also note that it is straightforward to extend M to cope with input goals which are neither unit nor variable-free.

The goals in the body of a clause may share variables. An alternating Turing machine cannot simulate solving a conjunctive goal with shared variables directly, as universally spawned processes do not share their tape. Hence M has to agree on the final value of the shared variables before universally invoking the processes that will work on each goal separately. M accomplishes this by choosing a substitution θ that both unifies the current goal with the invoked clause and instantiates all the goals in the body of the clause to their final values in the refutation R it simulates. The definition of M prevents it from further instantiating invoked goals, as the unification it performs is one-way (i.e. it checks that $A{=}A'\theta$, not that $A\theta{=}A'\theta$).

We say that *M accepts A in n iterations* if it has a computation tree that accepts A in which every path in the tree contains at most n occurrences of M being in its initial state.

Lemma 4.1: Let P be a logic program, M the ATM that simulates P as described in Simulation 1, A a variable-free goal, and τ_P the transformation associated with P as defined above. Then A is in $\tau_P^n(\{\})$ iff M accepts A in n iterations.

Proof: We prove the lemma by induction on n. For $n{=}1$, if A is in $\tau_P^1(\{\})$, then there is a clause $A'\leftarrow$ in P for which A is an instance via some substitution θ. Hence M can choose the clause $A'\leftarrow$ and the substitution θ, verify that $A{=}A'\theta$, and when it universally chooses a goal in the body of the clause $A'\leftarrow$ it accepts, since the body is empty.

Conversely, if M accepts A in one iteration, i.e., without returning to its initial state, it means that it had an empty universal choice during the first iteration. By the definition of M this can happen only if M found a clause $A'\leftarrow$ and a substitution θ such that $A{=}A'\theta$. By definition of τ_P, if follows that A is in $\tau_P^1(\{\})$.

Inductively assume that the lemma holds for any $i\leq n$, for some $n\geq 1$. If A is in $\tau_P^{n+1}(\{\})$, it follows by the definition of τ_P that there is a clause $p{=}A'\leftarrow B_1,B_2,\dots,B_k$, $k\geq 0$, and a substitution θ such that $A{=}A'\theta$ and $B_i\theta$ is in $\tau_P^n(\{\})$ for every $1\leq i\leq k$. By the inductive assumption, M accepts $B_i\theta$ in n iterations, for every $1\leq i\leq k$. Hence when applied to the goal A M can choose the clause p and the substitution θ, and accept A in $n{+}1$ iterations.

Conversely, assume that M accepts A in $n{+}1$ iterations. Let $p{=}A'\leftarrow B_1,B_2,\dots,B_k$ be the clause chosen by M in its first iteration, and θ the chosen substitution. M accepts $B_i\theta$ in n iterations, for every $1\leq i\leq k$. By the inductive assumption $B_i\theta$ is in $\tau_P^n(\{\})$, and by the definition of τ_P it follows that A are in $\tau_P^{n+1}(\{\})$. ▯

The following is a corollary of the lemma above and the fixpoint results of van Emden and Kowalski [9].

Corollary 4.2: Let P, M, and A be as in Lemma 4.1. Then M accepts A iff P solves A.

Simulation 1 describes the ATM M in high-level concepts. Before analyzing the complexity of M's computations we show how M can perform the necessary low-level computations and bookkeeping of each iteration in a reasonable amount of time and space, using three tapes. Each iteration begins with M having a variable-free unit goal A on its first tape. M then writes down on its second tape a substitution θ, and existentially chooses a clause $A' \leftarrow B_1, B_2, \ldots, B_k$ from P, which is stored in its finite control. It then computes $A'\theta$ on its third tape. If A' has c variables then M needs at most c passes on θ and no more than $|A'\theta|$ tape cells to compute $A'\theta$. It then verifies that $A = A'\theta$ by scanning its first and third tapes. Following this step M universally chooses a goal B_i, for some $1 \leq i \leq k$, and computes $B_i\theta$ on its first tape, using at most c' passes on θ, where c' is the number of variables in B_i. It then erases everything from its three tapes except for $B_i\theta$, and enters its initial state.

It is not difficult to see that the size of A and θ dominates the space and time needed for that iteration, and that if both are bounded by some constant g then M's iteration can be performed in space and time cg, for some constant c uniform in P. Furthermore, if θ is such that the size of $B_i\theta$ is bounded by some constant g', for all $1 \leq i \leq k$, there there is a substitution θ' such that $A\theta' = A\theta$ and $B_i\theta' = B_i\theta$, $1 \leq i \leq k$, and the size of θ' is bounded by $max\{g, kg'\}$. Hence the following lemma.

Lemma 4.3: Let P be a logic program and M the ATM that simulates P as defined in Simulation 1. Then there is a constant c uniform in P that bounds the complexity of M's iterations as follows. If M has a computation that accepts A in which the size of every goal is bounded by some $g > 0$ then M has a computation that accepts A, performs the same selection of clauses, operates in space cg and performs each iteration in time cg.

We proceed to analyze the complexity of M's simulations as a function of the complexity of P's refutations. We do so by showing that for any variable-free goal A and any refutation R of A from P there exists an accepting computation of M on A that mirrors R in a natural way, and bound the complexity of that computation.

Theorem 4.4: 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 alternating Turing machine M and a constant c uniform in P such that M operates in time $cD(n)G(n)$ space $cG(n)$ and tree size $cL(n)G(n)$, and that $L(M) = I(P)$.

Proof: Let $R = <N_0, C_0, \theta_0>, <N_1, C_1, \theta_1>, \ldots, <\Box, \{\}, \{\}>$ be a refutation of length l. The idea of M's computation that mirrors R is to make the same choices of clauses as R. When M invokes the clause $C_i = A \leftarrow B$ it applies to every goal B' in B a substitution θ. Since M does not change invoked goals, $B'\theta$ is the final instantiation of this goal in M's computation. Hence, in order for M to mirror the refutation R, M has to be clairvoyant about the final instantiation in R of the variables in B'. In other words, M has to choose θ such that $B'\theta = B'\theta_i\theta_{i+1}\ldots\theta_l$, for all

B' in B.

We argue that such a choice of θ does not impair ***M***'s ability to make the same choice of clauses as in ***R***. On the first goal of ***R***, $N_0=A$, ***M*** invokes the clause $C_0=A' \leftarrow B_1,B_2,...,B_k$ and chooses a substitution θ such that $B_i\theta=B_i\theta_0\theta_1...\theta_l$ for all i, $1 \leq i \leq k$. This provides the base case for our inductive argument.

Let A be a goal which is invoked in the i^{th} derivation step of ***R*** and is resolved in the j^{th} derivation step with the clause $C_j=A' \leftarrow B$ and the substitution θ_j. We can inductively assume that when ***M*** starts working on the goal A this goal is already instantiated to $A\theta$, where $\theta=\theta_i\theta_{i+1}...\theta_l$. By the definition of ***R***, θ_j is a unifier for $A\theta_i\theta_{i+1}...\theta_{j-1}$ and A'. Since $A\theta_i\theta_{i+1}...\theta_j=A'\theta_j$ it follows from properties of substitutions that $A\theta=A\theta_i\theta_{i+1}...\theta_j\theta_{j+1}...\theta_l= A'\theta_j\theta_{j+1}...\theta_l$. Hence $A\theta$ is unifiable with A', and ***M*** can choose the clause C_j.

Assume that the refutation ***R*** is of length l, goal-size g and depth d. We bound the space, time and tree-size of the computation of ***M*** that mirrors ***R*** as a function of l, g and d.

Consider the space of ***M***'s computation. By assumption the size of every goal in the computation is bounded by g, hence by Lemma 4.3 the space of ***M***'s computation need not exceed cg, for some constant c uniform in ***P***.

Consider the time of ***M***'s computation. Each of ***M***'s iterations corresponds to an invocation of a clause, hence the number of its iterations along any path in the accepting computation tree need not exceed d, the depth of ***R***. By Lemma 4.3 the time of each iteration need not exceed cg, for some constant c uniform in ***M***, hence the total time used by ***M*** is bounded by cdg.

Consider the tree size of ***M***'s computation. At most one universal branching occurs between two configurations in which ***M*** is in its initial state. The number of times ***M*** is in its initial state in the computation tree is bounded by l, the length of ***R***. The number of steps of each iteration is bounded by cg, hence the tree size of the computation is bounded by clg. Together these three claims establish the theorem. $\square$

4.2. Simulating an Alternating Turing Machine with a Logic Program

Naturally, to simulate existential branching in an ATM we use the nondeterministic choice of the clause to be invoked, and to simulate universal branching we use the goals in the body of the clause. Simulation 2 below describes a logic program ***P*** that simulates a one-tape alternating Turing machine ***M*** following these guidelines. It has one predicate *accept*(Q,L,R), with the property that for any configuration $<Q,L,R>$, ***P*** solves the goal *accept*(Q,L,R) iff this configuration leads to acceptance. The predicate stores in its first argument ***M***'s state, in its second argument the used part of the tape to the left of ***M***'s head, and in its third argument the used part of the tape to the right of ***M***'s head, including the cell ***M***'s head is positioned on. By "used part of the tape" we mean the smallest contiguous portion of the tape that includes all non-blank tape cells and all cells visited by ***M***.

Since the art of simulation is not as developed for logic programs as it is for Turing machines, the logic program that simulates the transitions of M is described explicitly. The program is slightly complicated by the need to treat reaching the ends of the used part of the tape as special cases.

Simulation 2: A logic program simulates a one-tape alternating Turing machine

Let M be a one-tape ATM. Its transitions are of the form $<q,\sigma,q',\tau,D>$, with the interpretation "from state q on symbol σ enter state q', write the symbol τ, and move in direction D". We define a logic program P that simulates M.

The simulating program P has one predicate, *accept*(L,Q,R), whose semantics is "the configuration $<Q,L,R>$ leads to acceptance". P has two types of axioms that define this semantics, which correspond to existential and universal configurations. We give some examples of such axioms; a complete description of them appears in Figure 3.

Axioms of the first type say that for any existential configuration α and configuration β that is a successor to α, α leads to acceptance if β leads to acceptance. For example, assume that M has the transition $<q,\sigma,q',\tau,left>$, where q is an existential state. The clause that corresponds to this transition says: "if α is a configuration with state q in which the head is positioned on the symbol σ, and the configuration β is a successor to α, obtained by writing τ, moving the head to the left, and entering state q', then α leads to acceptance if β leads to acceptance." The clause reads:

$accept(q,[X|L],[\sigma|R]) \leftarrow accept(q',L,[X,\tau|R]).$

This clause assumes that M's head is in the center of the tape, i.e., that neither the left half nor the right half of the tape are empty. The following clause handles the case in which the left half of the tape is empty:

$accept(q,[],[\sigma|R]) \leftarrow accept(q',[],[\#,\tau|R]).$

The head writes τ and moves to the left as before, but now it is positioned on the blank symbol $\#$, and the left half of the tape remains empty.

The case where the right half of the tape is empty needs treatment only if $\sigma{=}\#$, otherwise the transition is not applicable. The clause for this case is:

$accept(q,[X|L],[]) \leftarrow accept(q',L,[X,\tau]).$

After the head moves to the left, the right half of the tape contains τ, which is the symbol M wrote before it moved, and X, the symbol that was to the left of M' head.

Finally, if $\sigma{=}\#$ we also need a clause that deals with an empty tape:

$accept(q,[],[]) \leftarrow accept(q',[],[\#,\tau]).$

This clause combines ideas from the two previous clauses. The treatment of transitions in which the head movement is to the right is similar, except that it has fewer special cases.

The clauses of the second type correspond to universal configurations. They say that for any universal configuration α, if $\beta_1,\beta_2,..,\beta_k$, $k\geq 0$, are all the successors to α, then α leads to acceptance if β_1 and β_2 ... and β_k lead to acceptance; if $k=0$ the axiom simply says that α is accepting. To express this P has one clause for every pair $<q,\sigma>$ such that q is a universal state and σ is a tape symbol.

For example, assume that M has no transitions on the universal state q and the symbol σ. This means that a configuration in which M is in state q and looking at the symbol σ is accepting. The corresponding clause in P is:

accept$(q,L,[\sigma|R])$.

And if $\sigma=\#$, we also add to P the clause:

accept$(q,L,[])$.

As another example, assume that the only two transitions M has on the universal state q and the symbol σ are $<q,\sigma,p,\tau,right>$ and $<q,\sigma,p',\tau',left>$. The clause that corresponds to these two transitions is:

accept$(q,[X|L],[\sigma|R]) \leftarrow$ *accept*$(p,[\tau,X|L],R)$, *accept*$(p',L,[X,\tau'|R])$.

This transition assumes that the head is in the center of the tape. Similar to existential configurations, the cases where the head is in the left or right end of the tape need special treatment. For example, the clause that handles the case where the head has reached the left end of the tape is:

accept$(q,[],[\sigma|R]) \leftarrow$ *accept*$(p,[\tau],R)$, *accept*$(p',[],[\#,\tau'|R])$.

The treatment of the other cases is similar. Figure 3 summarizes them.

The generalization to a k-tape machine is not difficult: for each additional tape one adds to *accept* two arguments, for storing the left-half and right-half of the tape, and simulate the transitions accordingly. []

The correctness of Simulation 2 follows from a detailed, though simple, case analysis of the clauses in Figure 3, which shows that the refutation trees of the program P that simulates M reflect directly the complete computation trees of M. This analysis also shows that the depth complexity of P is identical to the time complexity of M, and that the length complexity of P is identical to the tree-size complexity of M. It is also easy to see that the goal-size of refutations for the program P that simulates M is linear in the space of M's computations, since each goal

- **Existential states.** For every existential state q and input symbol σ:
  - ▸ **Left move.** If M has a transition $<q,\sigma,q',\tau,left>$ then P has the clauses:
    - **Center of tape:** $accept(q,[X|L],[\sigma|R]) \leftarrow accept(q',L,[X,\tau|R])$.
    - **Left end of tape:** $accept(q,[],[\sigma|R]) \leftarrow accept(q',L,[\#,\tau|R])$.
    - **Right end of tape** (if $\sigma=\#$): $accept(q,[X|L],[]) \leftarrow accept(q',L,[X,\tau])$.
    - **Empty tape** (if $\sigma=\#$): $accept(q,[],[]) \leftarrow accept(q',[],[\#,\tau])$.
  - ▸ **Right move.** If M has a transition $<q,\sigma,q',\tau,right>$ then P has the clauses:
    - **Center and left end of tape:** $accept(q,L,[\sigma|R]) \leftarrow accept(q',[\tau|L],R)$
    - **Right end and empty tape** (if $\sigma=\#$): $accept(q,L,[]) \leftarrow accept(q',[\tau|L],[])$.
- **Universal states.** For every universal state q and input symbol σ, P contains clauses of the form $A \leftarrow A_1, A_2,\dots ,A_k$, where $k\geq 0$ is the number of transitions M has in state q on symbol σ.
  - ▸ **Center of tape:** $accept(q,[X|L],[\sigma|R]) \leftarrow A_1, A_2,\dots ,A_k$. If the i^{th} transition on $<q,\sigma>$ is $<q,\sigma,q',\tau,right>$ then A_i is the goal $accept([q',\tau,X|L],R)$. If that transition is $<q,\sigma,q',\tau,left>$ then A_i is $accept(q',L,[X,\tau|R])$.
  - ▸ **Left end of tape:** $accept(q,[],[\sigma|R]) \leftarrow A_1, A_2,\dots ,A_k$, where the goal for a transition $<q,\sigma,q',\tau,right>$ is $accept(q',[\tau],R)$ and for $<q,\sigma,q',\tau,left>$ is $accept(q',[],[\#,\tau|R])$.
  - ▸ **Right end of tape** (if $\sigma=\#$): $accept(q,[X|L],[]) \leftarrow A_1, A_2,\dots ,A_k$, where the goal for $<q,\#,q',\tau,right>$ is $accept([q',\tau,X|L],[])$ and for $<q,\#,q',\tau,left>$ is $accept(q',L,[X,\tau])$.
  - ▸ **Empty tape** (if $\sigma=\#$): $accept(q,[],[]) \leftarrow A_1, A_2,\dots ,A_k$, where the goal for $<q,\#,q',\tau,right>$ is $accept(q',[\tau],[])$ and for $<q,\#,q',\tau,left>$ is $accept(q',[],[\#,\tau])$. []

Figure 4: A logic program simulates a one-tape alternating Turing machine

in the computation is a notational variant of the corresponding configuration in the simulated computation.

Theorem 4.5: Let M be a k-tape alternating Turing machine that accepts 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 $T(n)$, goal-size complexity $cS(n)$ and length complexity $Z(n)$ such that $L(M)=\{X|\ accept(q0,[]^k,X,[]^{k-1})$ is in $I(P)\}$, where $q0$ is the initial state of M and c is a constant uniform in M. []

5. Applications

In this section we describe applications of the results above. They are based on the following observations concerning the logic program P that simulates the ATM M, as defined in Simulation 2.

1. If M does not go outside of its original input, then only clauses marked "center of tape" in Figure 3 need to be included in P. For any substitution, θ and any clause $A \leftarrow B_1,...B_k$ in P, the size of $B_i\theta$ is equal to the size of $A\theta$, $1 \leq i \leq k$.
2. If M is nondeterministic (i.e. with at most one transition per symbol in any universal state), then every clause of P contains at most one goal in its body.

Definition 5.1: A clause $A \leftarrow B_1,...,B_k$ is called *linear* if if for every i, $1 \leq i \leq k$, the size of B_i is less than or equal to the size of A, and the number of occurrences of any variable in B_i is less than or equal to the number of its occurrences in A.

Lemma 5.2: A linear logic program is of linear goal-size complexity.

Proof: (Informal) consider a refutation tree from such a program. The size of the sons in this tree can not exceed the size of their parent by the definition above. []

The following theorem characterizes Alternating Linear Space in terms of interpretations of linear logic programs.

Theorem 5.3: If P is a logic program of linear goal-size complexity then $I(P)$ is in Alternating Linear Space. If L is in Alternating Linear Space then there is a linear logic program P and a goal A containing the variable X such that $L = \{X\theta \mid A\theta \text{ is in } I(P)\}$.

Proof: Let P be a logic program of linear goal-size complexity. By Theorem 4.4 there is an ATM M such that $L(M) = I(P)$, and M operates in linear space. Hence $I(P)$ is in Alternating Linear Space.

Let L be a language in Alternating Linear Space. Then there is an ATM M such that $L(M) = L$ and M operates in linear space. By well-known compression techniques (cf. [13]) we may assume that M has only one tape, and that it does not go outside of the space of its original input. By Theorem 4.5 there is a logic program P such that P solves $accept(q0,[],X)$ iff X is in $L(M)$. Using the observations made above and the fact that M does not go outside of the space of its original input we can restrict P to contain only linear clauses. []

A clause $A \leftarrow B$, where B is a unit goal, is called a *transformation*. The following theorem characterizes Nondeterministic Linear Space in terms of interpretations of linear logic programs in which every clause is a transformation.

Theorem 5.4: If P is a logic program of linear goal-size complexity and every clause in P is a transformation then $I(P)$ is in Nondeterministic Linear Space. If L is in Nondeterministic Linear Space then there is a linear logic program P in which every clause is a transformation and an atom A containing the variable X such that $L = \{X\theta \mid A\theta \text{ is in } I(P)\}$.

Proof: Let P be a logic program of linear goal-size complexity such that every clause in P is a transformation. By Theorem 4.4 there is an ATM M such that $L(M)=I(P)$, and M operates in linear space. The use M makes of universal branching is to choose the goal in body of the invoked clause to work on next. Since P has at most one goal in its body, the corresponding M has only one universal choice in its only universal state. In other words, M is a nondeterministic Turing machine that operates in linear space. Hence $I(P)$ is in Nondeterministic Linear Space.

Let L be a language in Nondeterministic Linear Space. Then there is an ATM M such that $L(M)=L$, M operates in linear space, and does not go outside the space of its original input. By an argument similar to the proof above there is a linear logic program P such that P solves *accept*$(q0,[],X)$ iff X is in $L(M)$. Since M has at most one transition for any universal state q and tape symbol σ, the clauses in P are all transformations by construction. []

Corollary 5.5: Let P be a linear logic program in which every clause is a transformation. Then the problem of deciding whether P solves A, where A is a variable-free goal, is *PSPACE*-complete (cf. [13]).

6. Conclusions

After introducing the concept of alternation, Chandra et al. [3] comment: "Certain problems seem more convenient to program using the construct of alternation, but we do not know whether alternation will find its way into programming languages or have a role to play in structured programming. Such questions present themselves to further research". Motivated by the idea of applying alternation to structured programming, Harel [12] has developed And/Or programs. The results of this paper suggest that a programming language that embodies the concept of alternation already exists.

Logic programs are simple enough to be amenable to theoretical analysis and expressive enough to be a real programming language. This combination suggests that theoretical studies of this computational model are likely to have some practical implications, in addition to increasing our understanding of computing in general.

Acknowledgements

I am grateful to Dana Angluin and Gregory Sullivan for discussions that helped to establish the results described in this paper.

References

[1] K. R. Apt and M. H. van Emden.
Contributions to the Theory of Logic Programming.
Technical Report CS-80-12, Department of Computer Science, University of Waterloo, February, 1980.

[2] D. L. Bowen, L. Byrd, L. M. Pereira, F. C. N. Pereira and D. H. D. Warren.
PROLOG on the DECSystem—10 User's Manual.
Technical Report , Department of Artificial Intelligence, University of Edinburgh, October, 1981.

[3] A. K. Chandra, D. C. Kozen, L. J. Stockmeyer.
Alternation.
Journal of the ACM 28(1):114-133, January, 1981.

[4] C. L. Chang and R. C. T. Lee.
Symbolic Logic and Mechanical Theorem Proving.
Academic Press, New York, 1973.

[5] K. L. Clark and S. Gregory.
A relational language for parallel programming.
In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture.* ACM, October, 1981.

[6] A. Colmerauer.
Metamorphosis grammars.
In L. Bolc (editor), *Natural language communication with computers*, . Springer-Verlag, 1978.

[7] John S. Conery and Dennis F. Kibler.
Parallel interpretation of logic programs.
In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture.* Association for Computing Machinery, October, 1981.

[8] M. H. van Emden and G. J. de Lucena.
Predicate logic as a programming language for parallel programming.
In K. L. Clark and S. A. Tarnlund (editors), *Logic Programming*, . Academic Press, 1982.
To appear.

[9] M. H. van Emden and R. A. Kowalski.
The semantics of predicate logic as a programming language.
Journal of the ACM 23:733-742, October, 1976.

[10] M. J. Fischer and R. A. Ladner.
Propositional dynamic logic of regular programs.
Journal of Computer and System Sciences 18:194-211, 1979.

[11] C. Cordell Green.
Theorem proving by resolution as a basis for question answering.
In B. Meltzer and D. Michie (editors), *Machine Intelligence* 4, pages 183-205. Edinburgh University Press, Edinburgh, 1969.

[12] David Harel.
And/or programs: A new approach to structured programming.
ACM Transactions on Programming Languages and Systems 2(1):1-17, January, 1980.

[13] J. E. Hopcroft and L. D. Ullman.
Introduction to Automata Theory, Languages, and Computations.
Addison Wesley, 1979.

[14] Robert A. Kowalski.
Predicate logic as a programming language.
In *Information Processing* 74, pages 569-574. North-Holland, Amsterdam, 1974.

[15] Robert A. Kowalski.
Logic for Problem Solving.
Elsevier North Holland Inc., 1979.

[16] Robert A. Kowalski.
Algorithm = Logic + Control.
Communications of the ACM 22(7):424-436, July, 1979.

[17] Drew V. McDermott.
The Prolog phenomenon.
SIGART Newsletter 72, July, 1980.

[18] J. A. Robinson.
A machine oriented logic based on the resolution principle.
Journal of the ACM 12:23-41, January, 1965.

[19] P. Roussel.
Prolog: Manuel Reference et d'Utilisation.
Technical Report, Groupe d'Intelligence Artificielle, Marseille-Luminy, September, 1975.

[20] Walter L. Ruzzo.
Tree-size bounded alternation.
In *Proceedings of the 11th ACM Symposium on the Theory of Computing,* pages 352-359. Association for Computing Machinery, 1979.

[21] Warren D. H. D. , Pereira L. M. , Pereira F. C. N.
Prolog - the language and its imlementation compared with Lisp.
In *Symposium on Artificial Intelligence and programming Languages,* pages 109-115. SIGART/SIGPLAN, August, 1977.