ALTERNATION AND THE COMPUTATIONAL COMPLEXITY OF LOGIC PROGRAMS

EHUD Y. SHAPIRO

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 [19] 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 occurred 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, 15–17]), and the programming language Prolog [2, 20], which is based on this idea, is a viable alternative to the programming language Lisp in the domain of symbolic programming [18, 24].

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

*Address Correspondence to Ehud Y. Shapiro, Department of Applied Mathematics, Weizmann Institute of Science, Rehovot 76100, Israel.

52 Vanderbilt Ave., New York, NY 10017 0743-1066/84/$03.00
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 provides 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 the potential of logic programs as a programming language for parallel machines [5, 7, 8, 22, 23].

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

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

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.

An example of a logic program for quicksort is shown as Program 1. We use strings beginning with an upper-case character 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 result of unifying the term $[X,Y]X]$ with the list $[1,2,3,4]$ is $A = 1$, $B = 2$, $X = [3,4]$, and unifying $[X|Y]$ with $[a]$ results in $X = a$, $Y = []$.

**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).
```
PROGRAM 2. Simulating a pushdown automaton.

In establishing the relationship between computations of alternating Turing machines and logic programs, we develop below logic programs that simulate alternating Turing machines. A simpler precursor of these programs is Program 2. It is a logic program that simulates a two-state nondeterministic pushdown automaton that accepts palindromes over an arbitrary alphabet. The procedure \( \text{pal}(Q, X, Y) \) stores in \( Q \) the state of the automaton, in \( X \) the remaining input string and in \( Y \) the pushdown stack. The meaning of \( \text{pal}(Q, X, Y) \) is “The pda accepts the string \( X \) starting from state \( Q \) and stack contents \( Y \)”. The program is designed to succeed on the goal \( \text{pal}(q_0, S, []) \) iff \( S \) is a palindrome. The first four clauses in the program are of the form

\[
\text{pal}(q, [A|X], S) \leftarrow \text{pal}(q', X, S').
\]

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, \( \text{pal}(q_1, [], []) \), says “The pda accepts the empty string in state \(q_1\) and empty stack.”

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 \( A_1, A_2, \ldots, A_n \). A goal \( A_i \) and a clause \( A' \leftarrow B_1, B_2, \ldots, B_k \) in \( P \) are then chosen nondeterministically; the head of the clause \( A' \) is unified with \( A_i \) via a substitution \( \theta \), and the reduced goal is \( (A_1, \ldots, A_{i-1}, B_1, B_2, \ldots, B_k, A_{i+1}, \ldots, A_n)\theta \). 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 a possibility of confusion with other symbols (such as variables, integers). Variables are distinguished by an initial capital letter.

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 \( \theta = X_1 \rightarrow t_1, X_2 \rightarrow t_2, \ldots, X_n \rightarrow t_n \) 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 \( \theta \) 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 \( \theta_i \) of \( s_1 \) and \( s_2 \), \( s_1 \theta_i \) is an instance of \( s_1 \theta \). If two terms are unifiable then they have a unique (up to renaming variables) most general unifier [19].

We define computations of logic programs. Let \( N = A_1, A_2, \ldots, A_m, m \geq 0 \), be a (conjunctive) goal and \( C = A \leftarrow B_1, \ldots, B_k, k \geq 0 \), be a clause such that \( A \) and \( A_i \) are unifiable via a substitution \( \theta_i \) for some \( 1 \leq i \leq m \). Then \( N' = (A_1, \ldots, A_{i-1}, B_1, \ldots, B_k, A_{i+1}, \ldots, A_m) \theta \) is said to be derived from \( N \) and \( C \), with substitution \( \theta \). A goal \( A_i \theta \) 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 \( A_j \) 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 \( \langle N_i, C_i, \theta_i \rangle \), \( i = 0, 1, \ldots \) such that \( N_i \) is a goal, \( C_i \) is a clause in \( P \) with new variable symbols not occurring previously in the derivation, \( \theta_i \) is a substitution, \( N_0 = N \), and \( N_{i+1} \) is derived from \( N_i \) and \( C_i \) with substitution \( \theta_i \), for all \( i \geq 0 \).

A derivation of \( N \) from \( P \) is called a proof 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 proof of a goal \( A \) from a program \( P \) we also say that \( P \) solves \( A \).

Figure 1 shows a proof of the goal \( \text{pal}(q0, [a, b, a], []) \) from Program 2.

A more intuitive, though less complete way to describe a successful computation of a logic program (i.e., a proof) is via a proof tree. In a proof tree nodes are goals

---

**FIGURE 1.** An example of a proof.

\[
\langle \text{pal}(q0, [a, b, a], []), \text{pal}(q0, [A0|X0], Y0) \leftarrow \text{pal}(q0, X0, [A0|Y0]) \rangle,
\{ A0 \rightarrow a, X0 \rightarrow [b, a], Y0 \rightarrow [] \}
\]

\[
\langle \text{pal}(q0, [b, a, [a]], \text{pal}(q0, [A1|X1], Y1) \leftarrow \text{pal}(q1, X1, Y1),
\{ A1 \rightarrow b, X1 \rightarrow [a], Y1 \rightarrow [a] \}\rangle
\]

\[
\langle \text{pal}(q1, [a, [a]], \text{pal}(q1, [A2|X2], [A2|Y2]) \leftarrow \text{pal}(q1, X2, Y2),
\{ A2 \rightarrow a, X2 \rightarrow [], Y2 \rightarrow [] \}\rangle
\]

\[
\langle \square, \square, \{ \} \rangle
\]
FIGURE 2. An example of a proof tree.

that occur in the computation, with their variables instantiated to their final values, and arcs represent the relation of goal invocation. The proof tree that corresponds to the proof in Figure 2 is simply the list of goals in this proof, which are the first elements of the triples, connected with arcs. The proof tree in Figure 2 corresponds to the proof of \(qsort([2, 1, 3], L)\) from Program 1. Depth of indentation reflects depth in the 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 base 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 \(\{ A | A \in H(P) \}\) and \(P\) 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 \(\tau_p\) with any program \(P\), and show that \(I(P)\) is the least fixpoint of \(\tau_p\). The transformation \(\tau_p\) is defined as follows. Let \(I\) be a subset of \(H(P)\). Then a variable-free goal \(A \in H(P)\) is in \(\tau_p(I)\) iff there is a variable-free instance \(A \leftarrow B_1, B_2, \ldots, B_k\) of a clause in \(P\) such that \(B_i\) is in \(I\) for all \(i\).

### 2.4 Complexity Measures

We define complexity measures over proofs, using the notion of proof tree. Let \(R\) be a proof. We define the length of \(R\) to be the number of nodes in the proof 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 proof tree, where the size of a goal is the number of symbols in its textual representation.

Definition 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 proof 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 proof 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 proof 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 \( I(P) = I \) 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 if 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 if 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

\begin{align*}
Q & \text{ is the set of states,} \\
\Delta & \text{ is the input alphabet,} \\
\Gamma & \text{ is the tape alphabet,} \\
\# & \in \Gamma - \Delta \text{ is the blank symbol,} \\
\delta & \subseteq (Q \times \Gamma^k \times \{\text{left, right}\}^k) \text{ is the next move relation,} \\
q_0 & \in Q \text{ is the initial state,} \\
U & \subseteq Q \text{ is the set of universal states, and} \\
Q - U & \text{ is the set of existential states.}
\end{align*}

A nondeterministic Turing machine is an alternating Turing machine that 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 next move relation $\delta$.

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 $k$ 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 $\beta$ is a successor of a configuration $\alpha$ if $\beta$ follows from $\alpha$ in one step, according to the next move relation $\delta$. A computation path $\alpha_1, \alpha_2, \alpha_3, \ldots$ is a (possibly infinite) sequence of configurations of $M$ for which $\alpha_{i+1}$ is a successor of $\alpha_i$, for all $i \geq 1$.

A computation tree of $M$ is a rooted, directed tree whose nodes are configurations of $M$ which has the property that 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 $\alpha$ in $T$ and every successor $\beta$ to $\alpha$ there is an edge $(\alpha, \beta)$ in $T$.
2. All the leaves of $T$ are universal configurations.

A configuration $\alpha$ 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 $\langle q_0, [], x, [], k-1 \rangle$, 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 at most 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 at most $T(n)$ 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 at most $Z(n)$ that accepts $x$ (cf. [21]). Note that we measure only accepting computations.

4. SIMULATIONS AMONG ALTERNATING TURING MACHINES AND LOGIC PROGRAMS

The superficial 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 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.

One important difference is that conjunctive goals share variables, and hence cannot be solved independently, whereas the computations of universally spawned processes in an ATM are independent.

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. A key idea in the following simulation is that the unifying substitution, $\theta$, is chosen so that applying it to the chosen clause results in the body of that clause being ground.

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 $A' \leftarrow B_1, B_2, \ldots, B_k$, in $P$, and writes on its tape a substitution $\theta$. It then computes $A'\theta$, verifies that $A = A'\theta$, and erases everything from the tape except $\theta$. Then, using universal branching, it chooses $B_i$ for some $i$, applies $\theta$ to $B_i$, erases everything from the tape except $B_i\theta$, and returns to its initial state. $\square$

Note that $M$ accepts if the clause has an empty body, and rejects if it fails to find a clause in $P$ whose head unifies with $M$'s 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 $\theta$ 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 proof $R$ 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 contains at most $n$ occurrences of configurations in which $M$ is 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 $\tau_P$ the transformation associated with $P$ as defined above. Then $A$ is in $\tau_P^r(\{\})$ iff $M$ accepts $A$ in $n$ iterations.

[The Lemma is proven by induction on $n$.] 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 $A$ 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 $\theta$, 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 $\theta$ 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 $\theta$, 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 $\theta$ dominates the space and time needed for an 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 $c$ that depends on $P$. Furthermore, if $\theta$ is such that the size of $B_i\theta$ is bounded by some constant $g'$, for all $1 \leq i \leq k$, then there is a substitution $\theta'$ such that $A\theta' = A\theta$ and $B_i\theta' = B_i\theta$, $1 \leq i \leq k$, and the size of $\theta'$ 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 proofs. We do so by showing that for any variable-free goal $A$ and any proof $R$ of $A$ from $P$ there exists an accepting computation of $M$ on $A$ that mirrors $R$ in a natural way, and bounds 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 $R = \langle N_0, C_0, \theta_0 \rangle, \langle N_1, C_1, \theta_1 \rangle, \ldots, \langle \Sigma, \{\}, \{\} \rangle$ be a proof 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 \( \theta \). 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 proof \( R \), \( M \) has to be clairvoyant about the final instantiation in \( R \) of the variables in \( B' \). In other words, \( M \) has to choose \( \theta \) such that \( B'\theta = B'_1\theta_i\theta_{i+1} \ldots \theta_j \) for all \( B' \) in \( B \).

We argue that such a choice of \( \theta \) 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 \( C_0 = A' \leftarrow B_1, B_2, \ldots, B_k \), and chooses a substitution \( \theta \) such that \( B_i\theta = B_i\theta_0\theta_1 \ldots \theta_j \) 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 \( \theta_j \). We can inductively assume that when \( M \) starts working on the goal \( A \) this goal is already instantiated to \( A\theta \), where \( \theta = \theta_0\theta_1 \ldots \theta_j \). By the definition of \( R \), \( \theta_j \) is a unifier for \( A\theta, \theta_{j+1} \ldots \theta_{j-1} \) and \( A' \). Since \( A\theta, \theta_{j+1} \ldots \theta_{j-1} = A'\theta_j \), it follows from properties of substitutions that \( A\theta = A\theta_{j+1} \ldots \theta_{j+1} \ldots \theta_j = A'\theta_j \theta_{j+1} \ldots \theta_l \). Hence \( A\theta \) is unifiable with \( A' \), and \( M \) can choose the clause \( C_j \).

Assume that the proof \( R \) 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 \( \text{accept}(Q, L, R) \), with the property that for any configuration \( \langle Q, L, R \rangle \), \( P \) solves the goal \( \text{accept}(Q, L, R) \) 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 nonblank 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 \( \sigma \) enter state \( q' \), write the symbol \( \tau \), and move in direction \( D \)”. We define a logic program \( P \) that simulates \( M \).

The simulating program \( P \) has one predicate, accept\( (Q, L, R) \), whose meaning is “the configuration \( (Q, L, R) \) leads to acceptance”. \( P \) has two types of axioms that define these semantics, which correspond to existential and universal configurations. A complete description of them appears in Figure 3.

Axioms of the first type say that for any existential configuration \( \alpha \) and configuration \( \beta \) that is a successor to \( \alpha \), \( \alpha \) leads to acceptance if \( \beta \) leads to acceptance.

**FIGURE 3.** A logic program simulates a one-tape alternating Turing machine.

- **Existential states.** For every existential state \( q \) and input symbol \( \sigma \):
  - Left move. If \( M \) has a transition \( (q, \sigma, q', \tau, \text{left}) \) then \( P \) has the clauses:
    - Center of tape: \( \text{accept}(q, [X|L],[\sigma|R]) \leftarrow \text{accept}(q', L,[X, \tau|R]) \).
    - Left end of tape: \( \text{accept}(q,[\sigma|R]) \leftarrow \text{accept}(q',[,\#|R]) \).
    - Right end of tape (if \( \sigma = \# \)):
      - \( \text{accept}(q,[X|L],[]) \leftarrow \text{accept}(q', L,[X, \tau]) \).
    - Empty tape (if \( \sigma = \# \)):
      - \( \text{accept}(q,[],[],[\#, \tau]) \).
  - Right move. If \( M \) has a transition \( (q, \sigma, q', \tau, \text{right}) \) then \( P \) has the clauses:
    - Center and left end of tape: \( \text{accept}(q, L,[\sigma|R]) \leftarrow \text{accept}(q', [\tau|L], R) \)
    - Right end and empty tape (if \( \sigma = \# \)):
      - \( \text{accept}(q, [\sigma|R]) \leftarrow \text{accept}(q', [],[\#, \tau]) \).

- **Universal states.** For every universal state \( q \) and input symbol \( \sigma \), \( P \) contains clauses of the form \( A \leftarrow A_1, A_2, \ldots, A_k \), where \( k \geq 0 \) is the number of transitions \( M \) has in state \( q \) on symbol \( \sigma \).
  - Center of tape: \( \text{accept}(q,[X|L],[\sigma|R]) \leftarrow A_1, A_2, \ldots, A_k \). If the ith transition on \( (q, \sigma) \) is \( (q, \sigma, q', \tau, \text{right}) \) then \( A_i \) is the goal \( \text{accept}(q', [\tau|L], R) \). If that transition is \( (q, \sigma, q', \tau, \text{left}) \) then \( A_i \) is \( \text{accept}(q', L,[X, \tau|R]) \).
  - Left end of tape: \( \text{accept}(q,[\sigma|R]) \leftarrow A_1, A_2, \ldots, A_k \), where the goal for a transition \( (q, \sigma, q', \tau, \text{right}) \) is \( \text{accept}(q', [\tau|L], R) \) and for \( (q, \sigma, q', \tau, \text{left}) \) is \( \text{accept}(q', [],[\#, \tau|R]) \).
  - Right end of tape (if \( \sigma = \# \)):
    - \( \text{accept}(q,[X|L],[]) \leftarrow A_1, A_2, \ldots, A_k \), where the goal for \( (q, \#, q', \tau, \text{right}) \) is \( \text{accept}(q', [\tau, X|L],[]) \) and for \( (q, \#, q', \tau, \text{left}) \) is \( \text{accept}(q', L,[X, \tau]) \).
  - Empty tape (if \( \sigma = \# \)):
    - \( \text{accept}(q,[],[]) \leftarrow A_1, A_2, \ldots, A_k \), where the goal for \( (q, \#, q', \tau, \text{right}) \) is \( \text{accept}(q', [],[]) \) and for \( (q, \#, q', \tau, \text{left}) \) is \( \text{accept}(q', [],[\#, \tau]) \).
The clauses of the second type correspond to universal configurations. They say that for any universal configuration \( \alpha \), if \( \beta_1, \beta_2, \ldots, \beta_k, k \geq 0 \), are all the successors to \( \alpha \) then \( \alpha \) leads to acceptance if \( \beta_1 \) and \( \beta_2 \) and \( \beta_k \) lead to acceptance; if \( k = 0 \) the axiom simply says that \( \alpha \) is accepting. To express this \( P \) has one clause for every pair \( \langle q, \sigma \rangle \) such that \( q \) is a universal state and \( \sigma \) is a tape symbol.

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 simulates the transitions accordingly. \( \square \)

The correctness of Simulation 2 follows from a detailed, though simple, case analysis of the clauses in Figure 3, which shows that the proof 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 proofs for the program \( P \) that simulates \( M \) is linear in the space of \( M \)'s computations, since each goal 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 \( L(M) = \{ X \mid accept(q_0,[\mathcal{J}^k, X,[\mathcal{J}^{k-1}] \text{ is in } I(P)] \} \), where \( q_0 \) 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 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 \( \theta \) and any clause \( A \leftarrow B_1, \ldots, B_k \) in \( P \), the size of \( B_i \theta \) is equal to the size of \( A \theta \).

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 \( A \leftarrow B_1, \ldots, B_k \) is called linear 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 proof tree from such a program. The size of the sons in this tree cannot exceed the size of their parent by the definition above. \( \square \)

The following theorem characterizes Alternating Linear Space in terms of interpretations of linear 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 $L = \{ X\theta | A\theta \text{ is in } I(P) \}$.

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 $\text{accept}(q_0, [], X)$ 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 $L = \{ X\theta | A\theta \text{ is in } I(P) \}$.

Proof. Similar to theorem 5.3.

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 for 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 more likely to have some practical implications, in addition to increasing our understanding of computing in general.
I am grateful to Dana Angluin and Gregory Sullivan for discussions that helped to establish the results described in this paper. This work was carried out while the author was at Yale University and was supported by the National Science Foundation, grant No. MCS8002447.

REFERENCES

