The parallel complexity of single rule logic programs

Foto Afrati

Computer Science Division, Department of Electrical Engineering, National Technical University of Athens, 157 73 Zographou, Athens, Greece

Received 26 June 1990

Abstract


We consider logic programs without function symbols, called Datalog programs, and study their parallel complexity. We survey the tools developed for proving that there is a PRAM algorithm which computes the minimum model of a Datalog program in polylogarithmic parallel time using a polynomial number of processors (that is, for proving membership in \( \mathcal{R} \)). We extend certain of these tools to be applied to a wider class of programs; as they were, they were applied to chain rule programs (i.e., the relations on the right-hand side of the rule are binary and form a chain). We examine the parallel complexity of weak-chain rule programs (i.e., the relations on the right-hand side of the rule form a weak chain), and prove certain subclasses to belong to \( \mathcal{R} \). Finally we prove a wide class of programs to be log space complete for \( \mathcal{R} \) by giving sufficient conditions for a single rule program to be \( \mathcal{R} \) complete.

1. Introduction

A database query language that has received considerable attention recently is Datalog, the language of logic programs (known also as Horn clause programs) without function symbols and without negation [6, 8, 11, 14, 16]. Datalog programs or logical query programs essentially add recursion to first-order (algebraic) database query languages.

The following are two examples of logical query programs. The program \( \pi_1 \):

\[
S(x, y) : b(x, y),
\]

\[
S(x, y) := a(x, z_1), S(z_1, z_2), S(z_2, y),
\]
and the program $\pi_2$:

$$S(x, y) \quad := \quad b(x, y),$$
$$S(x, y) \quad := \quad a(x, z_1), S(z_1, z_2), S(z_2, z_3), a(z_3, y).$$

Each program consists of a set of Horn clauses called rules. For a certain instantiation of the variables of a rule to constants, the instantiated rule can be thought of as a deduction rule: “if the conjunction of the right-hand side predicates is true, then the left-hand side predicate is true”. A rule that uses $S$ (a left-hand side predicate) in the right-hand side is called recursive; $S$ is called a recursive predicate. In both examples, the second rule is recursive and the first is not. Consider a relation to be the interpretation of a predicate. Each of the above logical query programs, then, can be thought of as defining a function $Q$ from pairs of binary relations ($a$ and $b$) to binary relations ($S$, in the examples). Given the relations $a$ and $b$ (these comprise the extensional database $B$), the result of applying $Q$ on them is: the smallest binary relation $S$, such that, for all instantiations of all the rules, if the conjunction of the right-hand side predicates is true the left-hand side predicate is true too. This unique relation, $S$, is called the least fixpoint or the minimum model of the program on database $B$.

Datalog queries (that is, their minimum model) are computable in time that is polynomial in the size of the database $[2,5,10,12,19]$. The parallel complexity of Datalog programs is still not fully understood. Considerable recent research has addressed the problems of finding efficient methods and optimization techniques to compute Datalog queries $[4,15]$. Intelligent compilers is the prime goal of this research effort. One major direction of this area is to classify Datalog query programs as to their parallel complexity. Programs as alike in syntax as the two programs in the example turn out to belong to completely different parallel complexity classes. While $\pi_1$ is amenable to efficient parallelization (it belongs to the class $\mathcal{NC}$), program $\pi_2$ is inherently sequential (it is $\mathcal{NC}$-complete). $\mathcal{NC}$ is the class of problems that can be solved by a parallel random access machine (PRAM) in polylogarithmic time with polynomial number of processors $[13]$. Problems in $\mathcal{NC}$ are exactly those with a great deal of potential parallelism. The problems that are log space complete for $\mathcal{NC}$ are thought of as problems where considerable speed-ups cannot be achieved in parallel machines.

A parallel random access machine (PRAM), our computational model for parallelization, can be thought of as a collection of processors (RAM's) with random access to a common memory. We assume that concurrent reads are allowed and concurrent writes are allowed so far as they are consistent. We say that a problem belongs to the parallel complexity class $\mathcal{NC}$ $[13]$ iff there is a parallel algorithm that runs on a PRAM in polylogarithmic time ($O(\log^N)$) using a polynomial number of processors. As concerns membership to $\mathcal{NC}$, it does not matter what model we adopt with respect to concurrent reads and writes.

A logical query program is linear if the right-hand side of each rule involves at
most one recursive predicate. It is one of the earliest results that linear programs are in \( \mathcal{NC} \) [7,17]. Piecewise linear programs are a natural generalization. Perhaps they constituted the largest class of logic programs known to be in \( \mathcal{NC} \) up until recently [17]. It appears that in order to prove membership in \( \mathcal{NC} \) of other non-trivial classes of logic query programs we depend largely on the “polynomial fringe property” developed in [17] and the “polynomial stack property” developed in [1]. The next large classes of logic programs proved to belong to \( \mathcal{NC} \) are subclasses of chain rule programs (the predicates in the right-hand side of the rule are binary and they form a chain) [1]. Another class of single rule logical query programs that appears to have also nontrivial parallel complexity classification are weak-chain rule programs; they have also binary predicates in the right-hand side of the rule which form a chain but not necessarily a directed chain. At the other end, there is a large class of single rule programs with rather “complex” structure on the right-hand side of the rule, which can be proved to be log space complete for \( \mathcal{P} \).

The two main results of this paper address exactly these two points. First, we demonstrate a way for the polynomial stack property to be used for weak-chain rule programs too (as stated in [1] it applies to chain rule programs) and prove certain subclasses of weak-chain rule programs to belong to \( \mathcal{NC} \). Second, we state sufficient conditions on the syntax of the rule for a program to be \( \mathcal{P} \)-complete.

The remainder of this paper is organized as follows: In Section 2, we give basic concepts and definitions. In Section 3, we review the known results on the parallel complexity of logical query programs and we state the polynomial fringe property [17] and the polynomial stack property [1] theorems. In Section 4, we prove membership in \( \mathcal{NC} \) of subclasses of weak-chain programs. In Section 5, \( \mathcal{P} \)-completeness results are proved. Section 6 concludes the paper by discussing possible extensions of this research and presenting open problems.

2. Definitions

A database is a vector \( B = (D, r_1, r_2, \ldots, r_n) \), where \( D \) is a finite subset of a universe called the database domain, and \( r_i \subseteq D^{k_i} \) for some nonnegative integer \( k_i \); that is, \( r_i \) is a relation of arity \( k_i \) for each \( i = 1, \ldots, n \). An element of a relation is a tuple \( (a_1, \ldots, a_{k_i}) \), where \( a_j \in D \). We say that database \( B \) has sort \( (k_1, \ldots, k_n) \).

A query is a function \( Q \) from database of sort \( (k_1, \ldots, k_n) \) to database of sort \( (k) \) (i.e., to a single relation), such that \( Q(B) \subseteq D^k \), where \( B \) is a database with domain \( D \) and of sort \( (k_1, \ldots, k_n) \). In the case \( k = 0 \), we have a Boolean query which outputs either \( \{\} \) (false) or the one-element set containing the empty tuple, \( \{\} \) (true).

First-order queries are the queries expressible in first-order relational calculus. Fixpoint queries are obtained by augmenting first-order logic with the fixpoint operator. Logical queries are queries that can be expressed in the language of logical query programs, known as Datalog. The syntax and the semantics of logical query programs are given in the following.
Let $R_1, \ldots, R_n$ be extensional database (EDB) predicates, where the arity of $R_i$ is $\kappa_i$. Let $V$ be a countably infinite set of variables. An EDB atom has the form $R_i(x_1, \ldots, x_k)$, where $x_j \in V$. Let $\{P_1, P_2, \ldots\}$ be a countably infinite collection of intensional database (IDB) predicates, where the arity of $P_i$ is $\lambda_i$, and there are infinitely many predicates of every nonnegative arity. An IDB atom (of arity $\lambda_i$) has the form $P_i(x_1, \ldots, x_k)$, where $x_j \in V$. A Datalog program, or logical query program (we often abbreviate it to "logic program" or even "program" as that is the only kind we consider) $\pi$ is a collection of rules (function-free Horn clauses) of the form

$$l_0 := l_1, \ldots, l_k, \epsilon_1, \ldots, \epsilon_\kappa$$

where $\kappa, \lambda \geq 0$, the $\epsilon_m$ are IDB atoms, and the $\epsilon_m$ are EDB atoms. Atoms that appear on the right hand side of a rule are referred to as subgoals of the rule. The right-hand side of the rule is also called the body of the rule while the left-hand side is called the head of the rule.

Let $\pi$ be a logic program, as defined above, and let $B = (\mathcal{D}, R_1, \ldots, R_n)$ be a database, thought of as a collection of facts about the EDB predicates of $\pi$; i.e., if $(a_1, \ldots, a_k) \in R_i$ we say that $R_i(a_1, \ldots, a_k)$ is true, or that $R_i(a_1, \ldots, a_k)$ is a fact of the extensional database. A database $B$ of appropriate sort $(k_1, \ldots, k_n)$ (recall that $k_i$ is also the arity of predicate $R_i$ in $\pi$) is called an EDB instance of the logic program $\pi$. An extended logic program is the union of a logic program and an EDB instance. Now we give the semantics of an extended logic program. Let $I_0, I_1, \ldots$ be a sequence of databases, thought of as a collection of facts of both EDB and IDB predicates of program $\pi$. We define $I_0, I_1, \ldots$ inductively. $I_0$ contains the EDB facts of $B$ and all the IDB relations are empty; an IDB fact $P_i(a_1, \ldots, a_k)$ is contained in $I_k$ iff there is a rule, $r$, in $\pi$ and an instantiation of the variables of $r$ to constants in $D$ (i.e., a substitution by a constant of all occurrences of each variable) so that:

- the head of rule $r$ coincides with $P_i(a_1, \ldots, a_k)$ and all facts in the body of the rule are true in $I_{k-1}$.
- Observe that $I_{k-1} \subseteq I_k$ and $I_k$ contains the initial EDB facts plus the IDB facts that can be deduced from $B$ by at most $k$ applications of the rules.

It is easily shown that there is a finite integer $s$ such that $I_s = I_{s+1}$; we call $I_s$ the minimum model of the extended program. Database $I_s$ that contains deduced facts about the intentional predicates of the extended logic program is called the intensional database of program $\pi$. This operational semantics is equivalent to the fixpoint definition [3,18]: the minimum model of an extended logic program is the smallest database which satisfies all the rules (seeing as Horn clauses) of the logic program.

Let us distinguish an IDB predicate $P_i$ of program $\pi$. A logical query $Q_{\pi,P_i}$ on database (or EDB instance) $B$ is defined as a function that maps $B$ to a one-relation database $B'$. Database $B' = Q_{\pi,P_i}(B)$ has the same domain $\mathcal{D}$ as $B$ and the relation that comprises $B'$ contains exactly the facts of the IDB predicate $P_i$ in the minimum model.

The problem of computing the minimum model of a logic program $\pi$ on input $B$ is well known to be polynomial on the size of the input database $B$ (the program
\( \pi \) is considered fixed) [2, 5]. A polynomial algorithm (known as naive evaluation) is easily deduced from the above definition of the minimum model. Simply compute the sequence \( I_0, I_1, \ldots, I_s \). \( s \) cannot be larger than \( N^l \) where \( N \) is the size of \( B \) and \( l \) is the maximum arity of the predicates that appear in the rules of \( \pi \). Also, all the possible instantiations of the rules in each step (where \( I_k \) is computed from \( I_{k-1} \)) are not more than \( N^{\lambda \cdot l} \) where \( \lambda \) is the maximum length of the rules.

The data complexity problem of a logic program \( \pi \) is defined as follows [5, 19]: given an EDB instance \( B \), and a predicate \( P_i \) of \( \pi \) together with a tuple \( (c) \) of constants of appropriate arity, we ask the question "is \( P_i(c) \) a fact in the minimum model?". In other words, the data complexity problem is to decide membership of tuple \( (c) \) in the relation \( P_i \) (as corresponding to predicate \( P_i \)) of the minimum model.

Observe that there is an \( \mathcal{NC} \) algorithm for computing one step of naive evaluation: since there is a polynomial number of instantiations, we can consider all of them in parallel in at most polylogarithmic time (the exact time complexity depends on the specific model, but we are not going into this here). Clearly, for a fixed logic problem \( \pi \), the problem of computing the minimum model is in \( \mathcal{NC} \) iff the data complexity problem is in \( \mathcal{NC} \). Loosely speaking, we say that a logic program is in complexity class \( \mathcal{NC} \) or is \( \mathcal{P} \)-complete, when we really mean that the data complexity problem is. \( \mathcal{P} \)-complete problems are the problems that do not have an \( \mathcal{NC} \) algorithm unless \( \mathcal{NC} = \mathcal{P} \), a fact which is considered most unlikely.

We define a ground atom to be an atom with its variables instantiated to constants (e.g., a fact is expressed by a ground atom). An augmented database is defined to be a pair \((B, P(c))\) consisting of a database \( B = (D, r_1, \ldots, r_m) \) and a ground atom \( P(c) \), where \( c \) is a tuple of constants in \( D \). Consider a logic program \( \pi \) and a rule \( r \). We define the rule body of \( r \) to be an augmented database \((B, P(c))\) consisting of database \( B = (D, r_1, \ldots, r_m) \) and ground atom \( P(c) \), where: (a) The number of constants in \( D \) is equal to the number of distinct variables in the body of the rule and there is a one-to-one correspondence among them (thus, we conveniently think of the constant \( x' \) in \( D \) as corresponding to the variable \( x) \). (b) The relations \( r_i, i = 1, \ldots, m \) correspond in a one-to-one fashion to the predicates appearing in the body of the rule (again we think of the relation \( r_i \) as corresponding to the predicate \( R_i \), which may be either an EDB or IDB predicate). (c) A tuple of constants belongs to a relation \( r_i \) in \( B \) iff the corresponding tuple of variables constitutes the argument part of \( R_i \) in the body of the rule (see Fig. 1, where all the relations are binary and the rule body can be readily illustrated by a labeled graph). (d) The tuple \( (c) \) corresponds to the tuple of variables in the head of the rule and \( P \) is the predicate in the head of the rule.

A homomorphism from database \( B = (D, r_1, \ldots, r_m) \) to database \( B' = (D', r'_1, \ldots, r'_m) \) is a function \( h: D \rightarrow D' \) such that, if \((c_1, \ldots, c_k) \in r_i\), then \((h(c_1), \ldots, h(c_k)) \in r'_i\) (we do not require that all elements of \( D' \) are necessarily images of some element in \( D \)). A homomorphism from augmented database \((B, P(c))\) to an augmented database \((B', P(d))\) (let \( c = (c_1, \ldots, c_m) \) and \( d = (d_1, \ldots, d_m) \)) is a homomorphism \( h \) from \( B \) to
B' such that: \( h(c_i) = d_i, \ i = 1, \ldots, m \). A homomorphism from \((B, P(c))\) to \((B', P(d))\) is called total if for each element \( u \) in the domain of \( B' \) there is an element \( v \) in the domain of \( B \) such that \( h(u) = v \). If the homomorphism is total we call \((B', P(d))\) the homomorphic image of \((B, P(c))\). The following lemma is an immediate consequence of the definitions:

**Lemma 2.1.** Given a program \( \pi \) and a database \( B \), an IDB fact \( P_j(c_1, \ldots, c_m) \) is deduced in the minimum model iff there is an \( I_k \) in the sequence of databases \( I_0, \ldots, I_s \), defined above, and there is a rule \( r \) in \( \pi \) with rule body \((B_r, P_r(d))\) such that the following happens: There is a homomorphism \( h \) from the rule body of \( r \) to augmented database \((I_k, P_j(c_1, \ldots, c_m))\).

3. Polynomial fringes, polynomial stacks and \( A\forall \)

Consider an extended logic program consisting of a logic program \( \pi \) and an EDB instance \( B \). If a fact \( P_j(c) \) is true in the minimum model of \( \pi \) on \( B \), there is a computation that derives this fact. A derivation tree, \( T \), for \( P_j(c) \) describes a computation for \( P_j(c) \). Leaves of \( T \) are EDB facts, the root is \( P_j(c) \) and the internal nodes must satisfy the following: If nodes \( P_j(c_1), P_j(c_2), \ldots, P_j(c_j) \) are children of node \( P_0(c_0) \) in \( T \), there is a rule \( r \) in \( \pi \) and an instantiation of \( r \) by constants in \( D \), such that the instantiated rule is of the form:

\[
P_0(c_0) \ :- \ P_1(c_1), \ldots, P_j(c_j).
\]

Note that all nodes in a derivation tree belong to the minimum model. For each fact in the minimum model, there is always a derivation tree with polynomial (on
the size of the maximum path away from the root). This is easily deduced from the naive evaluation algorithm; just observe that if a fact \( P(c) \) is deduced after \( k \) applications of the rules (i.e., \( P(c) \in I_k \)), there is a derivation tree of depth \( k \). The fringe of a derivation tree is the collection of its leaves; the size of the fringe is the number of the leaves in the tree.

Ullman and Van Gelder [17] introduced the polynomial fringe property of a logic program. A logic program has the polynomial fringe property if for any EDB instance and for any fact in the minimum model of the resulting extended logic program there is a derivation tree whose fringe is of polynomial size in the size of the EDB instance. The polynomial fringe property decides membership in \( \mathcal{AC} \):

**Theorem 3.1** [17]. *If a logic program has the polynomial fringe property, it is in \( \mathcal{AC} \).*

We can prove that linear logic programs are in \( \mathcal{AC} \) using the polynomial fringe property. It is easily, also, proved from the polynomial fringe theorem. To define the next large class of \( \mathcal{AC} \) logic programs, we need some definitions [17]. Given a logic program \( \pi \), we construct the dependence graph of \( \pi \). The nodes of the graph represent predicate symbols (both EDB and IDB); there is an arc from node \( P_i \) to node \( P_j \) if there is a rule whose head is \( P_j \) and has a \( P_i \) subgoal. A rule in \( \pi \) is called recursive if it has a subgoal in the same strongly connected component as the head in the dependence graph. In a recursive rule, a recursive subgoal is a subgoal whose predicate symbol appears in the same strongly connected component as the predicate symbol in the head of the rule. Thus the same predicate may appear as recursive subgoal in one rule and as nonrecursive in another. Obviously, the strongly connected component of an EDB predicate consists of only one node. A logic program is called piecewise linear if each rule has at most one recursive subgoal. Piecewise linear programs belong to \( \mathcal{AC} \); this result was known before, it can be proved, though, using the polynomial fringe theorem, too [17].

Next, we consider a large class of logic programs, for which we have results concerning their parallel complexity. A chain program is a logic program in which all rules are such that: (a) All predicates are binary. (b) The variables appearing in the arguments of the predicate in the head of the rule are distinct and appear as the first argument of the first predicate and the last argument of the last predicate, respectively, in the body of the rule. (c) All other variables in the body of the rule are distinct, except that the last argument of the first predicate coincides with the first argument of the second predicate, the last argument of the second predicate coincides with the first argument of the third predicate, and so on. The programs in the example of the first section are chain programs. An EDB instance of a chain program can be viewed as a labeled directed graph; the labeled arcs define the binary relations of the input database. There is a natural way to associate with each chain program \( \pi \) a context-free grammar \( G(\pi) \). For example, with program \( \pi_1 \) of the Introduction, we associate grammar \( G(\pi_1) \):
While with program \( \pi_2 \) we associate grammar \( G(\pi_2) \):

\[
\begin{align*}
S & \rightarrow b, \\
S & \rightarrow aSS.
\end{align*}
\]

In general, to obtain \( G(\pi) \) from a chain program \( \pi \) we simply omit the variables and replace \( :- \) with \( \rightarrow \). Now, consider a chain program \( \pi \) and an EDB instance (i.e., a labeled graph) \( B \). It is easily deduced that an IDB fact \( P_i(c_1, c_2) \) is in the minimum model of \( \pi \) applied on \( B \) iff there is in \( B \) a directed path connecting \( c_1 \) to \( c_2 \) which spells (along its labeled arcs) a word of the language \( L(\pi) \) defined by the context-free grammar \( G(\pi) \). For any context-free language, there is a (nondeterministic) pushdown automaton that accepts it. We consider only standardized automata which (a) accept by empty stack and final state; (b) each move changes the stack by at most one symbol; and (c) the number of moves in an accepting computation is at least the length of the input word. (Any language generated by a context-free grammar with no empty rules can be generated by such an automaton. We say that a pushdown automaton has the polynomial stack property if for any computation of the automaton and for any integer \( h > 0 \), the number of different pushdown store contents of height \( h \) is polynomial on \( h \).

**Theorem 3.2** [1]. *If for a chain program \( \pi \) there is a pushdown automaton \( A \) accepting the language \( L(\pi) \), and \( A \) has the polynomial stack property, then program \( \pi \) has the polynomial fringe property.*

A simple logic program is a logic program with two rules, one of which consists of a single EDB predicate in its body (this is called the basis rule), and that predicate does not appear in the other rule which is a recursive rule. Again, programs \( \pi_1 \) and \( \pi_2 \) are both simple chain programs (\( b \) does not appear in the recursive rule). Hereafter, we shall refer to simple chain programs by giving only the recursive rule in the context-free grammar form (i.e., omitting the variables). In [1] a complete characterization of simple chain programs is given with respect to their parallel complexity:

**Theorem 3.3** [1]. (a) A simple chain program is in \( \mathcal{NC} \) if the recursive rule is linear or belongs to one of the following types:

\[
\begin{align*}
S & \rightarrow AS(BS)^j, \\
S & \rightarrow (SB)^jSA, \\
S & \rightarrow SAS(BSAS)^j, \\
S & \rightarrow (SA)^jSBS(AS)^j,
\end{align*}
\]
where \( i, j \) are nonnegative integers and \( A, B \) are strings of (possibly empty) EDB predicates.

(b) The simple chain programs that do not belong to one of the types of case (a) are \( \mathcal{P} \)-complete.

Remark. The notation \((A)^i\) means a string obtained by concatenating copies of the string \(A\), \(i\) times.

4. Simple weak-chain programs in \( \mathcal{NC} \)

Hereafter, we shall consider programs with binary predicates. Thus, rule bodies are labeled digraphs together with a distinguished pair of nodes. A path is a digraph with all nodes of in-degree (out-degree respectively) exactly one, except the beginning which has in-degree zero and the end which has out-degree zero. In terms of the rule body of a rule we can redefine chain rules: a rule with rule body being a directed path and distinguished nodes being the beginning and the end of the path respectively. A weak path is a digraph such that, if we ignore directions, we obtain an undirected path. A rule is called a weak-chain rule if the rule body is a weak path. We allow for any pair of nodes to be assigned as distinguished. A chain rule is a weak-chain rule. Also the following rules are weak-chain rules, but are not chain rules:

\[
S(x, y) \leftarrow a(y, z_1), S(z_1, z_2), S(z_2, z_3), a(z_3, x),
\]
\[
S(x, y) \leftarrow a(x, z_1), S(y, z_1),
\]
\[
S(x, y) \leftarrow a(x, z_1), c(z_3, z_1), S(z_2, z_3), S(z_2, z_4), c(z_4, y).
\]

We have shown how to associate chain rules with context-free grammars. We can establish a similar association for weak-chain rules. Consider the third rule of the above example. A simple program, that contains this rule as the recursive rule, can be rewritten as a chain program (but not simple any more), if we identify \(c^{-1}(x, y)\) with \(c(y, x)\), \(a^{-1}(x, y)\) with \(a(y, x)\) and \(S^{-1}(x, y)\) with \(S(y, x)\). The initial simple program \(\pi\) can, thus, be transformed to the following program \(\pi':\)

\[
S(x, y) \leftarrow a(x, z_1), c^{-1}(z_1, z_3), S^{-1}(z_3, z_2), S(z_2, z_4), c(z_4, y),
\]
\[
S^{-1}(x, y) \leftarrow c^{-1}(x, z_1), S^{-1}(z_1, z_3), S(z_3, z_2), c(z_2, z_4), a^{-1}(z_4, y),
\]
\[
S(x, y) \leftarrow b(x, y),
\]
\[
S^{-1}(x, y) \leftarrow b^{-1}(x, y).
\]

Programs \(\pi\) and \(\pi'\) are not equivalent in the sense that they give the same output on the same input. The output of \(\pi\) on database \(B\), though is the same with the output of \(\pi'\) on database \(B'\), where \(B'\) is obtained from \(B\) by adding the EDB relations
$b^{-1}$, $a^{-1}$, and $c^{-1}$ such as $b^{-1}(x,y)$ is an EDB fact if $b(y,x)$ is an EDB fact and similarly for relations $a$ and $c$. Hereafter, we refer to relation $b^{-1}$ as the reverse of relation $b$, and by extension, we refer to the symbol $b^{-1}$ as the reverse of the symbol $b$. We also define accordingly the reverse string $y^{-1}$ of a string $y$, where $y$ is consisting of both "simple" symbols and reverses of "simple" symbols. String $y^{-1}$ is obtained from string $y$ by spelling $y$ from the end towards the beginning and reversing the symbols (i.e., $b$ becomes $b^{-1}$ and $b^{-1}$ becomes $b$). For an example, if $y = aaba^{-1}cb^{-1}$, then $y^{-1} = bc^{-1}ab^{-1}a^{-1}a^{-1}$.

Bearing in mind the preceding remarks, program $\pi'$ (and, thus, program $\pi$ too) is completely described by the following context-free grammar rule:

$$S \rightarrow ac^{-1}S^{-1}Sc.$$  

We call this rule the type of the weak-chain program $\pi$. Moreover, if $S(u, v)$ is a fact of the minimum model of program $\pi$ on database $B$, there is a weak path, $p$, in $B$ from $u$ to $v$. Furthermore, if we redirect $p$ so that all arrows point from $u$ to $v$ and relabel the arcs that changed direction by the "reverse" of their label (e.g., we change label $a$ to $a^{-1}$), we can spell along path $p$ a word of the following context-free grammar $G(\pi)$:

$$ \begin{align*}
S & \rightarrow ac^{-1}S^{-1}Sc, \\
S^{-1} & \rightarrow c^{-1}S^{-1}Sca^{-1}, \\
S & \rightarrow b, \\
S^{-1} & \rightarrow b^{-1},
\end{align*}$$

where $a$, $b$, $c$, $a^{-1}$, $b^{-1}$, $c^{-1}$ are terminals [9].

In the rest of this section, we consider simple weak-chain programs; moreover, we assume that the two distinguished nodes of the rule body are the beginning and the end of the weak path respectively. We shall refer to them as "weak-chain programs" or simply "programs". Generalizing the preceding remarks, a simple weak-chain program $\pi$ associates with a context-free grammar $G(\pi)$ with two initialization rules and two recursive rules. The first recursive rule of $G(\pi)$ is obtained by considering the rule body of the only recursive rule of $\pi$ (let it be the weak path $p$ from $u$ to $v$) and spelling (reversing the symbol appropriately when traveling a reverse arc) the right-hand side of the rule following path $p$ from $u$ to $v$. The second rule is obtained similarly but spelling along $p$ from $v$ to $u$. From here on, we shall give only the one recursive rule of $G(\pi)$, in order to describe both program $\pi$ and the corresponding grammar $G(\pi)$.

Now, in order to apply the polynomial stack theorem to weak-chain programs too, we need to restate it as follows:

**Theorem 4.1.** Consider a weak-chain program $\pi$ and a context-free language $L$ with terminals corresponding to the EDB predicates of $\pi$ and their reverses. Assume that
$S(u, v)$ is a fact of $\pi$ on a database $B$ iff there is a weak path from $u$ to $v$ which spells a word of $L$. If there is a pushdown automaton $A$, which accepts $L$ and $A$ has the polynomial stack property, program $\pi$ is in JR?.

**Proof.** Since there is a pushdown automaton that accepts $L$, $L$ is a context-free language, therefore it is generated by a context-free grammar [9]. Let us associate with $L$ the chain program $\pi'$. $\pi'$ has twice as much predicates as $\pi$, i.e., it has the EDB predicates and IDB predicates of $\pi$ and their reverses (recall that, in the context of $L$ and $\pi'$, the "reverse of $a$" is a completely different symbol from $a$; we refer to it as "reverse" to illustrate its association with $a$ in the context of $\pi$). Program $\pi'$ is such that, for any database $B$, the extended program $(\pi, B)$ is equivalent to the extended program $(\pi', B')$; where $B'$ is obtained from $B$ by adding for each relation $r_j$ of $B$ one more relation $r_j'$ in $B'$ such that $(u, v) \in r_j'$ if $(v, u) \in r_j$. Because of Theorems 3.1 and 3.2, $(\pi', B')$ can be computed in $A\mathcal{P}$. \qed

Note that the language $L$ is not necessarily the language generated by the grammar $G(\pi)$.

**Theorem 4.2.** All simple weak-chain programs of type:

$$S \rightarrow (S^{-1})^i, \quad i=1,2,...$$

are in $A\mathcal{P}$.

**Proof.** Let $\pi$ be a weak-chain program of the above type for a specific $i=k$. Let database $B$ an EDB instance of $\pi$. Let $B'$ be the database obtained from $B$ by adding for each relation $r_j$ of $B$ a new relation $r_j'$ in $B'$ such that $(u, v) \in r_j'$ if $(v, u) \in r_j$. Since $B$ has one relation, let us call this relation $b$. Thus $B'$ has two relations, $b$ and $b^{-1}$. Consider a string $t$ of $b$'s and $b^{-1}$'s; we call the effective length of $t$ (and denoted $l(t)$) the number of $b$'s in $t$ minus the number of $b^{-1}$'s in $t$. For example, the effective length of $t = b^{-1}b b^{-1}b b b b^2 b^{-1}b^{-1}b^{-1}b^{-1}$ is $-2$.

We shall prove that, given the program $\pi$ and the database $B'$, $S(u, v)$ is a fact iff there is a path in $B'$ from $u$ to $v$ spelling a word $w$ such that: (a) The effective length of $w$ is $(k+1)i - k$ for any integer $i$ and (b) there is a substring of $w$ identical to $(b^{-1})^k$.

The "only iff" direction is an easy induction. We shall prove the "iff" direction by induction on the length of the path. In fact, we shall prove a slightly stronger result: Consider the intentional database $I_s$ (with two relations, $b$ and $S$) and obtain accordingly database $I_s'$ (with four relations $b$, $S$, $b^{-1}$, and $S^{-1}$). Define accordingly the effective length of a string of $S$'s and $S^{-1}$'s. We shall prove that $S(u, v)$ is a fact iff there is a path in $I_s'$ from $u$ to $v$ spelling a word $w$ such that: (a) The effective length of $w$ is $(k+1)i - k$ for any integer $i$ and (b) there is a substring of $w$ identical to $(S^{-1})^k$. 


For \( w \) with length equal to one, the inductive assertion is trivially true. Suppose it is true for any word of length \(< \lambda \). Let \( w \) be a word of length \( \lambda \) which is spelled along path \( p \). By hypothesis, \( w \) can be written: \( w = w_1(S^{-1})^k w_2 \). We shall show that there is also a shorter path from \( u \) to \( v \) which satisfies the assertion. We have five cases, depending on the last symbol of \( w_1 \) and the first symbol of \( w_2 \):

**Case a:** \( w = w_1(S^{-1})^k w_2 \). If there are more than \( 2k \) consecutive \( S^{-1} \)'s in \( w \), we substitute \( k \) of them by \( S \) (i.e., by applying the rule) and we get a shorter path. Otherwise \( w \) can be written: \( w = w_1 S(S^{-1})^{k+1} w_2 \) (or "symmetrically", with similar treatment). Observe that substring \( w' = S(S^{-1})^k \) implies a path from the beginning of the first symbol to the end of the third which spells \( S(S^{-1})^k S^{-2} \). Thus we can substitute \( w' \) for \( S^{-1} \). Doing this substitution in \( w \) we get a shorter string. It remains to be considered the case when one of the \( w_i \)'s is empty; this is Case e below.

**Case b:** \( w = w_1 S(S^{-1})^{k+1} w_2 \). The same substitution as in Case a works.

**Case c:** \( w = w_1 S(S^{-1})^{k} w_2 \). Symmetrical to Case b.

**Case d:** \( w = w_1 S(S^{-1})^{k+1} w_2 \). All subcases are treated similarly as in Case a except when \( w = w_1 S(S^{-1})^{j} S(S^{-1})^j S_{w_2} \) with \( 2 \leq j \). In this subcase consider the substring \( S^{-1} S(j(S^{-1})^k) \); from the beginning of this substring to somewhere (the exact place depends on \( j \)) in \( (S^{-1})^k \) there is a path that spells \( S(S^{-1})^k \). Thus, according to the remark in Case a, we can substitute \( S^{-1} S(j(S^{-1})^k) \) by \( S^{-1}(S^{-1})^k \).

**Case e:** \( w = (S^{-1})^k w_2 \). (The case \( w = w_1 S^{-1} \) is symmetrical.) The arguments are similar to the ones in Cases a and d.

Let \( \gamma \) be a string of symbols. We call \( \gamma \) a palindrome if there is a string \( \beta \) such that \( \gamma = \beta \beta^{-1} \). Let \( \gamma \) be any string and let \( \sigma \) be a string consisting of \( S \)'s and \( S^{-1} \)'s. We denote by \( \gamma \sigma \gamma \) the string obtained if we insert \( \gamma \) in every place between two consecutive symbols of \( \sigma \). For example if \( \sigma = SSS^{-1} \) and \( \gamma = ac^{-1}ca^{-1} \), then \( \sigma \gamma = Sac^{-1}ca^{-1}Sac^{-1}ca^{-1}S^{-1} \). The following theorem is a consequence of Theorems 2.1 and 4.2:

**Theorem 4.3.** Let \( \gamma \) be a string of EDB predicates which is a palindrome, and let \( \sigma \) be a string of \( S \)'s and \( S^{-1} \)'s, where \( S \) is the recursive predicate. Let \( \sigma \) be such that the rule body of rule

\[
S \to (S^{-1})^{k+j+1}
\]

is a homomorphic image of the rule body of rule

\[
S \to (S^{-1})^{k} \sigma(S^{-1})^{j}.
\]

Then, all simple weak-chain programs of type

\[
S \to ((S^{-1})^{k} \sigma(S^{-1})^{j})^{\gamma}
\]

are in \( \mathcal{N}W \).
Proof. First, we shall prove that any program $\pi$ of type

$S \to (S^{-1})^k \sigma(S^{-1})^l$

where $\sigma$ is such that the hypothesis of the theorem is satisfied, is in $\mathcal{AC}$. In the same sense as in Theorem 4.2, we shall construct a database $B'$ from input database $B$. We shall show that $S(u,v)$ is an IDB fact iff there is a path in $B'$ from $u$ to $v$ spelling a word $w$ such that: (a) The effective length of $w$ is $(k+i+2)I_i - (k+i+1)$ for any integer $i$ and (b) there is a substring in $w$ identical to $(b^{-1})^k \sigma'(b^{-1})^l$ (where $\sigma'$ is obtained from $\sigma$ by replacing any occurrence of $S$ by $b$ and of $S'$ by $b^{-1}$).

Just observe that the existence of a path in $B'$ spelling $w' = (S^{-1})^k \sigma(S^{-1})^l$ implies also the existence of a path (from the beginning to the end of $w$) spelling $w'' = (S^{-1})^k \sigma(S^{-1})^l S^{-1}(S^{-1})^k \sigma(S^{-1})^l$. The next important observation is: If we add a new rule $r'$ to the set of the rules of program $\pi$ such that the rule body of $r'$ is a homomorphic image of a rule $r$ in $\pi$, then the obtained program $\pi'$ is equivalent to $\pi$. Thus, let $w_1$ be the string obtained from $w$ by replacing $w'$ by $w''$. Hereafter, we can follow the proof of Theorem 4.2, by applying on $w_1$ the rule

$S \to (S^{-1})^{k+I+1}$

Finally, consider a program $\pi_1$ of type

$S \to ((S^{-1})^k \sigma(S^{-1})^l)^\gamma$

Observe that a path that deduces a fact of $\pi_1$ spells a word $w$ such that $w = (w')^\gamma$ and $w'$ is a word that deduces a fact for the program of type

$S \to (S^{-1})^k \sigma(S^{-1})^l$.

Other weak-chain programs known to be in $\mathcal{AC}$ are of the following two types:

$S \to SS^{-1}$

and

$S \to SaS^{-1}$.

For the first program a path deduces a fact iff it spells a word that begins with a $b$ and ends with a $b^{-1}$. For the second program a path deduces a fact iff it spells an odd-length word beginning with $b$ and which has $b$'s and $b^{-1}$'s in the odd positions and $a$'s and $a^{-1}$'s in the even positions [17]. It seems plausible that the methods of Theorems 4.2 and 4.3 can be used to prove wider classes of weak-chain programs in $\mathcal{AC}$. For example, one can relax the condition of Theorem 4.3 and allow the rule body to have a homomorphic image of type $(S^{-1})^k$ with any $\lambda$ (in Theorem 4.3, $\lambda = k + I + 1$).

5. $\mathcal{P}$-complete programs

Before we state the main theorem of this section we need some technical defini-
Consider a rule \( r \) with rule body \((B, P(c))\). Suppose there is a homomorphic image, \((B', P(d))\), of \((B, P(c))\) and a subgraph, \( H \), of \( B \) which includes all nodes appearing in tuple \( (c) \) (a subgraph of graph \( G \) is consisting of a subset of the arcs of \( G \)) such that the following happen: There exists an isomorphism \( \text{iso} \), from \( B' \) to \( H \) such that \( \text{iso}(d_i) = c_i \) for all \( i = 1, 2, \ldots \). Then, we say that rule \( r \) is in nonminimal form; otherwise, we say that \( r \) is in minimal form.

Taking into account Lemma 2.1, it is easily deduced that, for any program \( \pi \) there is an equivalent program \( \pi' \) with all rules in minimal form.

Some more definitions follow. Let \((B, P(c))\) be the rule body of a recursive rule; recall that \( B \) is a labeled digraph and \( P \) is a binary predicate. Let \( u \) be a node of \( B \); we denote by \((a)\) \( L_u^+ \) the set of labels of the outgoing arcs from \( u \) with a superscript \( '+' \) and by \((b)\) \( L_u^- \) the set of labels of the ingoing arcs in \( u \) with a superscript \( '-' \) (e.g., in Fig. 1(b), \( L_{s_1}^+ = \{a^+\} \) and \( L_{t_2}^- = \{c^-\} \)). The set \( L_u \) is the union of sets \( L_u^+ \) and \( L_u^- \). Suppose \( B \) is a directed acyclic graph (dag) with one source, \( s \), and one sink, \( t \) (a source, in a graph, has no ingoing arcs and a sink has no outgoing arcs). The length of a path in \( B \) is the number of arcs in the path. A minimum path from a node \( u \) to a node \( v \) is a path of minimum length. The distance between two nodes is the length of the minimum path connecting them. Consider all paths from \( s \) to \( t \) that include arc \( e \). We define the position of arc \( e \) in \( B \) as the length of the minimum such path from \( s \) to \( t \). Let \( p_{\text{max}} \) denote the maximum ever appearing position in \( B \). Finally a digraph is called disconnected if there is at least a pair of nodes for which there is no weak path connecting them. A cut of a digraph is a set of arcs which disconnects the digraph if deleted.

**Theorem 5.1.** Let \( \pi \) be a simple program with recursive rule \( r \) and corresponding rule body \((B, S(s, t))\) in minimal form. Suppose:

(a) \( B \) is a labeled directed acyclic graph (dag) with one source, \( s \), and one sink, \( t \). Moreover, no homomorphic image of \( B \) is isomorphic to an acyclic digraph which has a cut consisting solely of \( S \)-labeled arcs.

(b) There are two arcs \((s_I, t_I)\) and \((s_{II}, t_{II})\) labeled \( S \) in \( B \), called \( S_I \) and \( S_{II} \) respectively such that (i) the six nodes \( s, t, s_I, t_I, s_{II}, t_{II} \) are all distinct nodes, (ii) no other arc adjacent to either of these six nodes is labeled by \( S \), (iii) the position of \( S_I \) (\( S_{II} \) respectively) in \( B \) is equal to \( p_{\text{max}} \) and (iv) the distance of \( t_I \) from \( t \) equals the distance of \( t_{II} \) from \( t \).

(c) \( L_s \cap (L_{s_I} \cup L_{s_{II}}) = \emptyset \), \( L_t \cap (L_{t_I} \cup L_{t_{II}}) = \emptyset \).

(d) For any node \( v \) of \( B \), \( L_s \) is no subset of \( L_v \) and \( L_t \) is no subset of \( L_v \).

Then \( \pi \) is \( \mathcal{P} \)-complete.

**Proof.** We shall describe a logarithmic space reduction of the *Monotone Circuit Value Problem* to any program which satisfies conditions (a)-(d). In this problem we are given a Boolean circuit. A circuit consists of a directed acyclic graph, whose nodes are called *gates* and are divided into three categories: (a) The *input gates*, which have in-degree zero (and are further subdivided into those that have value one
and those that have value zero), (b) the AND gates and (c) the OR gates, all of indegree two. One of the gates is designated to be the output gate. The problem is to determine the value of the output gate, under the obvious computational rules for the AND and OR gates. The problem remains \( \mathcal{P} \)-complete even if we assume that all gates have out-degree two or less, and furthermore that any gate which has out-degree two appears once as the “left” input of a gate, and once as the “right” input of another gate [17].

Given any such circuit \( C \), we shall construct one relation for each EDB predicate and one question of the form “is \( S(c, c') \) an IDB fact?” (\( c, c' \) constants) such that the answer to the question is yes if and only if the output of \( C \) is one. Variants of this construction, which is of a form first used in [17], are used for the most \( \mathcal{P} \)-completeness proofs of logic programs that we know of, such as in [1].

For each gate \( x \) in \( C \), our database has two constants \( x_0, x_n \), and a number of other constants, depending on the kind of the gate, and the length of an expanded form of the rule to be defined below. The binary relations are defined in a way that reflects the structure of both the program and \( C \). Finally, we can show that gate \( x \) has value one if and only if \( S(x_0, x_n) \) is a fact and thus the answer to the query \( S(o_0, o_n) \)? with \( o \) the output gate yields the value computed by the circuit.

Let us first expand the rule body \( (B, S(s, t)) \) in the following fashion. First obtain \( B' \) from \( B \) by substituting all labels \( S \) by \( b \) except from the label on the arcs \( S_I \) and \( S_H \); also obtain \( B'_I \) from \( B' \) by replacing the \( S \) label on the \( S_I \) arc by \( b \) and, obtain \( B'_H \) from \( B' \) by replacing the \( S \) label on the \( S_H \) arc by \( b \). Consider \( B' \) and consider arc \( S_I \) in \( B' \), delete arc \( S_I \) and replace it in \( B' \) by a dag isomorphic to \( B'_I \) coinciding \( s \) with \( S_I \) and coinciding \( t \) with \( t_I \). In fact, we construct a sequence of databases \( B_0, B_1, \ldots, B_i, \ldots, B_2i \), such that \( B_0 = B' \), \( B_1 \) is the database just described and in general: (a) For \( j = 1, 2, \ldots, i \), \( B_j \) is obtained from \( B_{j-1} \) by considering the most recently put \( B'_I \)-isomorphic dag, considering its \( S_I \) arc, and substituting a new \( B'_I \)-isomorphic dag in the place of this \( S_I \) arc and (b) for \( j = i+1, \ldots, 2i \), \( B_j \) is obtained from \( B_{j-1} \) in exactly the same way except that a \( B'_I \)-isomorphic dag is used and arc \( S_H \) is considered for replacement (instead of arc \( S_I \)). For an example, see Fig. 2, where \( i = 2 \).

In order to be able to refer to them, we shall define \( 4i + 6 \) special nodes of digraph \( B_{2j} \). Consider \( B_j \) for \( j = 1, 2, \ldots, i \); in \( B_j \), there are only two arcs labeled by \( S \), an \( S_I \) arc and an \( S_H \) arc. Call the endpoints of the \( S_I \) arc \( s_j^I \) and \( t_j^I \) (the arc is directed from \( s_j^I \) to \( t_j^I \)). The nodes \( s_j^I \) and \( t_j^I \), \( j = 1, 2, \ldots, i \) are special nodes. Consider \( B_j \) for \( j = i+1, \ldots, 2i \), and denote by \( s_j^H \) and \( t_j^H \) (where \( j'-j-i \)) the endpoints of the \( S_H \) arc. The nodes \( s_j^H \) and \( t_j^H \), \( j' = 1, 2, \ldots, i \) are special nodes. Finally, the nodes \( s \) and \( t \) (the source and the sink) of \( B_0 \) and the four endpoints of the \( S \)-labeled arcs in \( B_0 \) are special nodes too. Integer \( i \) is chosen such that all possible distances between any two of these six nodes is greater than \( p_{\text{max}} \). Finally, let \( V_0 \) be the set of nodes of \( B_0 \), and let \( V_j \) be the set of nodes that belong to \( B_j \) but do not belong to \( B_{j-1} \).

Consider database \( B_{2j} \); let \( n-1 \) be the number of nodes that it contains. If \( x \) is an input gate, then two new constants, \( x_0 \) and \( x_n \), are added. If \( x \) is an AND gate,
then we add constants $x_0$ and $x_n$ and also we add $n - 1$ new constants $x_1, \ldots, x_{n-1}$. If $x$ is an OR gate, then we add constants $x_0$ and $x_n$ and we also add two sets of $n - 1$ constants $x_1, \ldots, x_{n-1}$ and $x_{n+1}, \ldots, x_{2n-1}$. Next, we construct the relations as follows: For an input gate $x$ of value one we add to relation $b$ (the nonrecursive rule) the pair $(x_0, x_n)$. If $x$ is an AND gate, we identify each constant $x_i$, $i = 0, \ldots, n$ with a node in $B_{2i}$, such as $x_0$ is identified to $s$ and $x_n$ to $t$. For each arc labeled $l$ (any $l$ except $S$) and with endpoints identified to constants $x_{i1}$ and $x_{i2}$ we add to the relation $l$ the pair $(x_{i1}, x_{i2})$. For an OR gate, we do the same for the $x'_i$ too (only that we think of $x'_0$ as identical to $x_0$ and of $x'_{n}$ as identical to $x_{n}$).

Notice that, so far, we have not taken into account the interconnections between gates of $C$. We introduce in the database the structure of $C$ as follows: $S$ appears twice in $B_{2i}$, and, in particular once appears as a $S_f$ arc and once as a $S_f'$ arc; let $x_{f1}, x_{f2}$ and $x_{f11}, x_{f12}$ be the constants corresponded to the endpoints of these arcs respectively. Then, if $x$ is an AND gate with left input $y$ and right input $z$, we identify $x_{f1}$ with $y_0$, $x_{f2}$ with $y_n$, $x_{f11}$ with $z_0$, and $x_{f12}$ with $z_n$ (see Fig. 2 for an example). If $x$ is an OR gate with left input $y$ and right input $z$, we identify $x_{f1}$ with $y_0$,
$x_{12}$ with $y$, $x_{12}'$ with $z_0$, and $x_{12}$ with $z_0$, and finally add to relation $b$ the pairs $b(x_{11}',x_{12})$ and $b(x_{11}',x_{12})$. This completes the construction of the database. We claim that, for any gate $x$, $S(x_o,x_i)$ is true if and only if the value computed by $x$ in $C$ is one.

We first prove the if direction. Suppose that the value computed by $x$ is one; we shall show by induction on the level of gate $x$ in $C$ that $S(x_o,x_i)$ is true. For the basis, if $x$ is an input gate, there is nothing to prove, since $b(x_o,x_i)$. If $x$ is an AND gate of value one, then both of its inputs $y$ and $z$ have also value one. By induction, $S(y_o,y_i)$ and $S(z_o,z_i)$ are true. It follows that, by reversing the expansion that produced $B_{2i}$, $S(x_o,x_i)$ is derived. For an OR gate $x$, either its right or its left input is one. Suppose that its left input $y$ is one (the argument for the right input $z$ is similar, only with the constants $x_j$ instead of the $x_i$). By induction, $S(z_o,z_i)$, and thus, again by reversing the expansion that produced $B_{2i}$, $S(x_o,x_i)$ is derived.

The only if direction is quite a bit more complicated. The result follows from an inductive assertion, stated and proved below. Recall the sequence of databases $I_0, I_1, ..., I_i$ that defines the minimum model of a logic program. The induction will be done on this sequence. Before stating the induction, we introduce some notation. Recall the special nodes and sets of nodes defined on $B_{2i}$. We denote by $x(s_i')$ the constant created by (AND or OR) gate $x$ and was corresponded to node $s_i$ of $B_{2i}$. Accordingly, we call $X(V_j)$ the set of constants corresponded to the set $V_j$ of $B_{2i}$. The inductive assertion consists of the following clauses:

**Inductive assertion.**

1. For any AND gate $x$, if $S(x_o,x_i) \in I_k$, then $S(x_v(s_i'),x_v(t_i')) \in I_{k-1}$ and $S(x_v(s_i'),x_v(t_i')) \in I_{k-1}$. Also, for any OR gate $x$, if $S(x_o,x_i) \in I_k$, then either $S(x_v(s_i'),x_v(t_i')) \in I_{k-1}$ and $S(x_v(s_i'),x_v(t_i')) \in I_{k-1}$ or $S(\hat{x}(s_i'),x_v(t_i')) \in I_{k-1}$ and $S(\hat{x}(s_i'),x_v(t_i')) \in I_{k-1}$ or both.

2. For any AND or OR gate $x$ and $j = 0, 1, ..., i-1$ the following happens: If $S(x_v(s_i'),z) \in I_k$, then (a) for $j = 1, ..., i-1$ either $z = x(t_i')$ and $S(x_v(s_i'),x_v(t_i')) \in I_{k-1}$ or $z = w(t_i')$ for some $r$ or $z = w(t_i')$ for some $r$ and (b) for $j = 0$, $z = x(t_i')$ and $S(x_v(s_i'),x_v(t_i')) \in I_{k-1}$.

3. We can distinguish the nodes mentioned in clauses (1)–(3) as $s$-special nodes and $t$-special nodes, in the obvious way. For any other pair of nodes $u$ and $v$, in
the input database, such that either \( u \) is different from any \( s \)-special node or \( u \) is different from any \( t \)-special node, if \( S(u, v) \in I_k \) then \( b(u, v) \in I_0 \).

**Proof.** For \( k = 0 \), there is nothing to prove.

(1) Suppose \( x \) is an AND gate and \( S(x_0, x_n) \in I_k \). According to Lemma 2.1, there is a homomorphism from the rule body of the recursive rule \( r \), to the augmented database \( (I_{k-1}, S(x_0, x_n)) \). Observe that, as a consequence of the four clauses of the assertion, \( I_{k-1} \) is \( I_0 \) enhanced with some (or none) additional \( S \)-arcs as follows: (a) arcs of the kind listed in clauses (1)-(4) and (b) (perhaps) some more \( S \)-arcs which, though, lead necessarily from \( s \)-special nodes to \( t \)-special nodes (because of condition (d) of Theorem 5.1). Let \( Im \) be the set of nodes of \( I_k \) that are images of some node in the rule body according to this homomorphism. Note that, as a consequence, for any \( v \in Im \) there must exist a path from \( x_0 \) to \( x_n \) through \( v \) of length at most \( p_{\text{max}} \). Consider any node \( u \) in \( I_k \) not in the set \( X(V, \cdot) \). We shall show that \( u \) is not a member of \( Im \). Because of conditions (b) (ii) and (c) of the theorem, all images adjacent to \( x_0 \) and \( x_n \) are in \( X(V_0) \). Thus, other nodes (besides the ones in \( X(V_0) \)) can be reached from \( s \) either over \( x(s_y) \) or \( x(s_y^\prime) \) and they can reach \( t \) either over \( x(t_y) \) or \( x(t_y^\prime) \). In any case the minimum path from \( s \) to \( t \) over \( u \) has length greater than \( p_{\text{max}} \) (because of conditions (b) (iii), (b) (iv)). Thus, \( u \) is not a member of \( Im \). Now, among the nodes of \( X(V_0) \) there are the EDB arcs, and, perhaps, the two \( S \)-labeled arcs (from \( x(s_y) \) to \( x(t_y) \) and from \( x(s_y^\prime) \) to \( x(t_y^\prime) \)) according to clause (2). Thus, the only possible homomorphism (since the rule is in minimal form) is the obvious isomorphism, which implies that \( S(x(s_y), x(t_y)) \in I_{k-1} \) and \( S(x(s_y^\prime), x(t_y^\prime)) \in I_{k-1} \).

The proof for the OR gate is similar.

(2) Similarly, if \( S(x(s_y), z) \in I_k \), there must exist a homomorphism from the rule body to the augmented database \( (I_{k-1}, S(x(s_y), z)) \). Because of condition (d), \( z \) equals to some \( t \)-special node. If \( z = x(t_y) \), by similar argumentation as in clause (1), we show that the only homomorphism is the obvious isomorphism and thus \( S(x(s_y^{j+1}), x(t_y^{j+1})) \in I_{k-1} \). As a consequence of the inductive clauses (2) and (3), \( z \) might be some other \( t \)-special node as well for \( j = 1, 2, \ldots, i-1 \). For \( j = 0 \), though, since \( i \) is chosen large enough, if \( z \neq x(t_y^0) \), then (a) the homomorphic image is acyclic, because no path reaches as far as \( y_0 \) and (b) there is a nonempty set of nodes in \( Im \) that are reached from \( x(s_y) \) only over an \( S \)-labeled arc. This means that the homomorphic image is acyclic and has an only-\( S \)-labeled cut; this contradicts condition (a).

(3) Similar to clause (2) for \( j \neq 0 \).

(4) It is a trivial consequence of condition (d) of the theorem.

Once the inductive assertion is proved, the only if part of the validity of the construction follows. At the final stage, clause (1) implies that \( S(x_0, x_n) \) if and only if gate \( x \) has value one. Thus, if \( o \) is the output gate, \( S(0_0, o_n) \) if and only if \( C \) computes the value one, and hence the rule was shown \( \mathcal{P} \)-complete. □

The conditions in Theorem 5.1 can be relaxed in a large extent; e.g., in [1], much "simpler" in structure programs are proved \( \mathcal{P} \)-complete. The difficulty is that we do
not know a general argument which can prove $\mathcal{P}$ completeness for a large class of programs, though they can be proved $\mathcal{P}$-complete with specialized arguments (as it was the case in [1], where a number of cases were considered).

6. Discussion and open problems

We have addressed the problem of the parallel complexity of logical query programs. We have surveyed certain tools for proving membership to $\mathcal{N}^c$ and have used the polynomial fringe property and the polynomial stack property to prove a subclass of weak-chain programs in $\mathcal{N}^c$. The next step would be to find the parallel complexity of uniform weak-chain programs (a uniform program has no EDB predicates in the body of the recursive rule; uniform simple chain programs are trivial) and then, uniform simple single rule programs in general. Our conjecture is that uniform weak-chain programs belong to $\mathcal{N}^c$ and probably even general uniform simple programs belong to $\mathcal{N}^c$ too. Apart from this, very few programs seem to belong to $\mathcal{N}^c$: one interesting class, still to consider for membership in $\mathcal{N}^c$, are simple programs with rule body that has a "simple" homomorphic image (e.g., corresponding to a rule body of an $\mathcal{N}^c$ program). For more general programs, the extend to which this technique can be used is limited by the fact that it is undecidable whether a general chain program has the polynomial fringe property [17].

At the other end, we have given a paradigm $\mathcal{P}$-completeness reduction for single rule logic programs. This reduction has been also used to prove all $\mathcal{P}$-completeness results in [1], and we believe that more subclasses of logic programs can be proved $\mathcal{P}$-complete by using it. The constraints in Theorem 5.1 can be relaxed and may be replaced by weaker ones. An interesting subclass to be considered for $\mathcal{P}$-completeness is the class of simple programs with rule body which has only nontrivial (by trivial we mean, for example, a weak chain) homomorphic images and with condition (d) in Theorem 5.1 relaxed.

References