A number of algebras, calculi, and rule-based query languages have been developed in recent years to meet the challenges of modern applications. Of these, rule-based languages are of interest to researchers in the areas of both databases and logic programming. This paper analyzes the "power" of a certain class of rule-based languages, called value-based languages. By "power" we mean data complexity. The main result of the paper is the establishment of both the upper and lower bounds of the data complexity of the finite versions of three (value-based) logic programming languages: ELPS, COL, and LDL. An interesting consequence of our analysis is a new technique to extend a given total order on a set to its power set using positive rules only (including for built-in predicates).

1. INTRODUCTION

Modern applications such as CAD, CAM, or scheduling require storing and reasoning with complex data containing nested sets and tuples. In response to this need, different kinds of database programming languages have been developed: (i) higher-order calculi/algebras [1, 16, 19, 24]; (ii) rule-based languages extending Datalog with set-manipulating primitives [2, 3, 9, 13, 17, 21, 22, 26]; (iii) functional languages [8, 11, 27]; (iv) extensions of SQL [7, 20]. Of these languages, the rule-based languages are of interest to researchers in the areas of both database querying and logic programming. Rule-based languages can be further divided into those that use object identity as a language construct (the reference-based languages) and those that use only nested sets and tuples (the value-based languages). Although the "power" of the reference-based languages and various calculi/algebras has been analyzed...
[4, 16, 17], similar analysis for value-based languages is not reported. This paper analyzes the “power” of the value-based languages ELPS, LDL, and COL.

By “power” we mean data complexity [12, 28]. The notion of data complexity is well motivated. The data complexity of a query \( \Phi \) is the complexity of determining whether a given database \( D \) entails \( \Phi \). This notion extends naturally from simple queries to programs written in the value-based languages. It gives an estimate of the complexity of answering queries as a function of database size. It does not take into account the size of the query, and because in most cases query size is negligible compared to database size, this is justified. Data complexity can be used to measure either time or space bounds. In this paper we investigate both the upper and lower time bounds of the data complexity of the value-based languages proposed in the literature.

The languages we consider are finite-ELPS, finite-LDL, and COL [2]. The first two are typed and function-free versions of ELPS [22] and LDL [9], respectively. “Function-free” means that the languages disallow function symbols; “typed” means the predicates are allowed to take nested tuples of some type (see below) specified a priori. Functions are disallowed for two main reasons: (i) in theory, data complexity becomes uninteresting when function symbols are added—it is known that data complexity of even Datalog is Turing-complete if function symbols are added; (ii) in practice, these finite versions are sufficient for databases with nested tuples and sets; furthermore, they play, in the context of nested-object data models, the same role as played by Datalog in the context of the relational data model.

We introduce (informally) three notions that are useful to describe our results: type, level, and exponential hierarchy. The first notion, the type of a predicate, determines what sort of data the predicate can contain. For example, if the type of a predicate is \( \langle D, \{D\} \rangle \), then it means that the first attribute of \( P \) can take atomic values only, whereas the second can take set values only. An example of a predicate whose type allows sets of sets is the predicate \( Q \) with a typing given by \( \langle D, \{\{D\}\} \rangle \); the second attribute of \( Q \) can have sets of sets only. Using the notion of type, we now define the second notion: level of a predicate. The level of a predicate determines the maximum depth of set-nesting of any of its attributes. Thus, for example, if the type of a predicate \( P \) is \( \langle D, \{D\} \rangle \), then its level is 1 because the attribute with maximum depth of nesting (the second attribute) has only one level of nesting.

On the other hand, the level of \( Q \) defined above is 2. A \( k \)-level language is a collection of programs that use predicates with a level of at most \( k \). In case of \( k \)-level COL, the levels of the data functions as well as the predicates are restricted. Finally, the third notion, exponential hierarchy, is a sequence of complexity classes \( \text{PTIME}, \text{EXPTIME}, \ldots \), where each member is formed by “an exponentiation” of the previous one in a manner formalized later (from [16]). We denote these classes by \( E_0, E_1, \ldots \), where \( E_0 \) denotes \( \text{PTIME} \).

The main result of the paper consists of first identifying a set of crucial constructs of the three value-based languages, and then (using these constructs) obtaining both the lower and upper bounds of the data complexity of all three languages. Specifically, we show that the data complexity of the \( k \)-level finite-ELPS, finite-LDL and COL is bounded from below (lower bound) and above (upper bound) by the \( k \)th level of the exponential hierarchy. We would like to point out that the lower bounds are obtained by using only the positive fragments of the languages. In fact, we do not require negation even on the built-in predicates (i.e., “=” and “\( \in \)”) to obtain the lower bounds. There are two interesting aspects to this work: (i) the importance of the upper and lower bounds per se and (ii) the techniques used to obtain them.

Regarding (i) above, our results are different from what one would expect from similar
such results—from the fact [14, 16] that higher order calculi (such as those of [16]) can simulate fixed points, one would expect (as in [5]) that our $k$-level programs would have the same data complexity as $\text{CALC}_k$ of [16]. However, this turns out not to be the case. For instance, $\text{CALC}_k$ does not have tight upper and lower bounds (see [16] for details), whereas we exhibit tight bounds for data complexity of our $k$-level languages.

Regarding (ii) above, the techniques we use are different from those used for similar results in the literature. The crucial part of any data complexity result involves simulating an arbitrary Turing machine in the specific complexity class. This, in turn, involves obtaining a total order of requisite length that can be used to count the steps of the Turing machine that is being simulated. Note that one can supply an order of length equal to the size of the database $|\mathcal{D}|$ through the input to the simulating program. However, for simulating Turing machines arbitrarily high in the exponential hierarchy, one needs to extend the given order of length $n$ to an order of length $2^n$ using the constructs of the language. This procedure can then be applied repeatedly to obtain the desired length arbitrarily high in the exponential hierarchy. Our approach to extending the domain order is different from the approaches followed in the literature (such as [10, 16, 23]). Note that [10] uses alternating PSPACE machines to obtain an EXPTIME data complexity result (and hence does not need to extend the domain order to exponential length), whereas [23] and [16] use such constructs as "$*$" and "negation" which we do not have access to in our language (recall that (i) our lower bounds do not use negation and (ii) the languages we study do not have "$*$").

Furthermore, the techniques we use yield two interesting “by-products”: (i) The power of grouping: We show that the finite-LDL, with the help of its grouping rule, can express nonmonotonic queries such as $\text{EQUAL}$ and $\text{PARITY}$. Since grouping is a nonmonotonic operator in itself, this may not appear surprising. However, it is known [6] that at least one of $\text{PARITY}$ or $\text{EQUAL}$ cannot be expressed by a number of nonmonotonic query languages such as stratified-Datalog, fixpoint queries, bounded-loop queries, etc. (ii) The interplay between positive rules with domain order and nonmonotonicity: We show that with domain order, even positive rules encode a number of nonmonotonic operations. In particular, we show that the positive finite-ELPS can $\text{group}$ (in the sense of [9]) sets that have a domain order on their members. This is in contrast with the well-known result [22] that positive ELPS cannot simulate (assuming no order on the domain) the grouping construct.

The rest of the paper is organized as follows: Section 2 introduces the three languages through examples and shows how to encode some basic (and crucial) set operations in these languages. Section 3 introduces the basic notions used in the later sections. Section 4 obtains the “by-products” mentioned above. Section 5 obtains the main result that the data complexity of a $k$-level language is bounded (above and below) by the $k$th level of exponential hierarchy. This result is obtained by simulating Turing machines running in the exponential hierarchy using orders of sufficient length. This section assumes that the orders required are available a priori. These simulations are fairly intuitive, and are provided in full detail to make the paper self-contained. Section 6 shows how to obtain the orders of sufficient length that are crucial for the encodings in Section 5 for simulating the Turing machine computations. Thus, this section provides the technical machinery required for establishing the main results of the paper. The last section concludes the paper.

2. A BRIEF INTRODUCTION TO COL, LDL, AND ELPS

In this section we introduce the three languages: ELPS [22], LDL [9], and COL [2]. We give sufficient details about the syntax and semantics of these languages to make this a
self-contained report. An interested reader is referred to the respective original papers for full formal syntax and semantics. Our primary aim here is to illustrate and introduce the languages, and show how they can encode some very useful set operations: union and disjoint union.

2.1. ELPS

Kuper’s ELPS extends Horn clause logic by a construct called restricted universal quantifier. A typical rule involving this construct is of the form

\[ A \leftarrow (\forall x_1 \in X_1) \cdots (\forall x_n \in X_n) [B_1 \land \cdots \land B_m]. \]

It means that if for every member of \( X_1, \ldots, X_n \), the body of the rule (namely, \([B_1 \land \cdots \land B_m]\)) holds, then conclude the head. It was proved in [22] that ELPS has as minimal model property, model intersection property, and unique answer property (all of which are enjoyed by Horn clause logic programs [25]). ELPS can be used to describe a number of interesting set operations [22].

Example 2.1. (Set operations in ELPS). If \( X \) and \( Y \) are set variables and \( x \) and \( y \) are atomic, then

\[
\text{DISJOINT}(X, Y) \leftarrow (\forall x \in X)(\forall y \in Y)(x \neq y)
\]

produces pairs of sets that are disjoint. If \( X \) and \( Y \) are set variables and \( x \) is an atomic,

\[
\text{SUBSET}(X, Y) \leftarrow (\forall x \in X)(x \in Y)
\]

computes pairs of sets such that the first one is a subset of the second. Computing the union of two sets is more complicated in ELPS, and the following rule achieves this:

\[
\begin{align*}
\text{temp}(z, X, Y) & \leftarrow z \in X \\
\text{temp}(z, X, Y) & \leftarrow z \in Y \\
S_o.U(Z, X, Y) & \leftarrow (\forall z \in Z) [\text{temp}(z, X, Y)] \\
\text{UNION}(X, Y, Z) & \leftarrow \text{SUBSET}(X, Z), \text{SUBSET}(Y, Z), \\
S_o.U(Z, X, Y).
\end{align*}
\]

In the above, the predicate \( \text{temp}(z, X, Y) \) holds if \( z \in X \) or \( z \in Y \). Clearly, then, the predicate \( S_o.U(Z, X, Y) \) holds for any \( Z \subset X \cup Y \) (\( Z \) is a Subset of the Union of \( X \) and \( Y \)). Finally, the last rule tests if \( Z \) is a subset of \( X \cup Y \) and it also contains both \( X \) and \( Y \). Indeed \( Z = X \cup Y \). The following rule computes the disjoint union:

\[
\text{DU}(X, Y, Z) \leftarrow \text{UNION}(X, Y, Z), \text{DISJOINT}(X, Y)
\]

The restricted ELPS that we consider is obtained from ELPS by disallowing function symbols and making the predicates “typed,” as we defined in Section 3.

2.2. LDL

LDL [9] extends Horn clause logic by adding a new construct called grouping. This is added specifically for the capability to build sets.

Example 2.2. (Grouping). Let \( \text{Teach}(t, c) \) stand for the fact that the teacher \( t \) teaches a course \( c \). Then the query get all the courses taught by “John” is answered by the following program involving grouping:
ON THE POWER OF RULE-BASED LANGUAGES

GROUP BY(t, {c}*) ← Teach(t, c)
Output(S) ← GROUP BY(t, S), (t = 'John').

In the above, the first rule (called grouping rule) collects, for each teacher t, the set of courses associated with t through the relation Teach; this set term is denoted by {c}*. Thus, for example, if Teach(John, CS540), Teach(John, CS541), and Teach(John, CS542) are all and only tuples of Teach with “John” in the first argument, then the grouping rule produces the tuple GROUP BY(John, {CS540, CS541, CS542}). The second rule projects the set of all courses offered by “John.”

The first rule, called grouping rule, produces a set of arbitrary size; by that we mean the size of the set produced is dependent on the database, not just on the program. This is unlike Datalog where the size of the tuple produced can be determined from the maximum arity of the predicates used in the program. We use a notation {x} to indicate grouping, as opposed to < x > used by [9]. We do this to indicate explicitly the fact that the size of the set produced by a grouping construct can be of arbitrary size, including “empty set” (denoted by Ø). A number of set operations can be encoded using grouping in a natural way [9].

Example 2.3. (Set operations using grouping). The operation of “union” of two sets is computed as follows:

\[ temp(X, Y, z) ← (z ∈ X) \]
\[ temp(X, Y, z) ← (z ∈ Y) \]
\[ UNION(X, Y, [z]*) ← temp(X, Y, z). \]

The operation of “intersection” of two sets is computed as follows:

\[ temp(X, Y, z) ← (z ∈ X), (z ∈ Y) \]
\[ INTERSECTION(X, Y, [z]*) ← temp(X, Y, z). \]

The operation of “disjoint union” is obtained as follows:

\[ DU(X, Y, Z) ← UNION(X, Y, Z), INTERSECTION(X, Y, Ø). \]

2.3. COL

COL [2] extends Datalog with “data functions.” These functions can be viewed as “predicates parameterized on variable names.” The following illustrates this.

Example 2.4. (Data functions—Grouping). Let Teach(t, c) stand for the fact that the teacher t teaches the course c. Then the query get all the courses taught by “John” is answered as follows:

\[ Data._Func(x) ∨ y ← Teach(x, y) \]
\[ GROUP BY(x, Data._Func(x)) ← Teach(x, y) \]
\[ Output(S) ← GROUP BY('John', S). \]

The first rule groups or nests the y-values associated with a given x into a set denoted by the data function “Data._Func(x)”, and the second rule evaluates or dereferences those values into the second attribute of the predicate “GROUP BY.” Thus, if Teach(John, CS540), Teach(John, CS541), and Teach(John, CS542) are the only tuples of Teach with “John” as the first attribute, then the first rule produces
Data(Func(John)) whose value is the set \{CS540, CS541, CS542\}. The second rule then produces a tuple Group_By(John, \{CS540, CS541, CS542\}) whereas the third projects the desired value.

Data functions can be used to perform a number of set operations as the following example will illustrate.

Example 2.5. (Set operations). The following rules compute “union,” “intersection,” and “disjoint union” of sets:

\[
\begin{align*}
U_1(X, Y) & \ni z \leftarrow (z \in X) \\
U_1(X, Y) & \ni z \leftarrow (z \in Y) \\
UNION(X, Y, Z) & \leftarrow U_1(X, Y) = Z.
\end{align*}
\]

In the above, \(U_1(X, Y)\) is a data function that collects the appropriate values, and these values are dereferenced in the third attribute of the predicate \(UNION\):

\[
\begin{align*}
I_1(X, Y) & \ni z \leftarrow (z \in X), (z \in Y) \\
INTERSECTION(X, Y, Z) & \leftarrow I_1(X, Y) = Z \\
DU(X, Y, Z) & \leftarrow UNION(X, Y, Z), INTERSECTION(X, Y, \emptyset).
\end{align*}
\]

Note that we considered only two operations—union and disjoint union—because these constitute the most useful programming idioms for these languages.

3. BASIC NOTIONS

In order to develop the main results of the paper, we need to define certain notions. First we formalize the notions of typing, finite versions, and “levels.”

Definition 3.1. (Type). The type expression \(D\) stands for a base type, such as \textbf{Char}. If \(\tau\) is a type expression, then \(\{\tau\}\), called set type, is also a type expression. Similarly, if \(\tau_1, \ldots, \tau_k\) are type expressions, then \((\tau_1, \ldots, \tau_k)\), called tuple expression, is also a type expression.

The following defines the “finite versions” of LDL and ELPS that we consider in our paper.

Definition 3.2. (Finite Versions). Finite-ELPS is obtained from ELPS [22] by (i) restricting it to be function-free and (ii) requiring that every predicate is associated with a type expression, called type of the predicate. Finite-LDL is obtained from LDL [9] similarly.

Next, we define the depth of set nesting of a type expression and then define “level” of a predicate. We use this notion to define \(k\)-level languages such as \(k\)-level finite-ELPS, \(k\)-level finite-LDL, etc.

Definition 3.3. (Level). The depth of nesting of \(D\) is zero, that of \(\{\tau\}\) is one more than that of \(\tau\), and that of \((\tau_1, \ldots, \tau_k)\) is the maximum of the depths of \(\tau_1\) through \(\tau_k\). The level of a predicate \(P\) is the depth of nesting of the type of \(P\). The level of a program is the maximum level of any predicate in the program and a \(k\)-level language is a collection of \(k\)-level programs.

An important aspect of all the three papers (ELPS [22], LDL [9], and COL [2]) is in establishing \textit{intended model semantics}. Each of the papers establishes that given a program
in the respective language, there is a unique intended model associated with that program. In the case of positive ELPS, [22] shows that the intended model is the minimal model obtained by applying the “T-operator” analogous to Horn clause logic programs [25]. In the case of LDL [9], however, even positive programs do not satisfy the unique minimal model property nor the model intersection property. The authors define a certain ordering on predicates, extend it to atoms and then to models, and define the notion of an intended model. A similar analysis is performed in the case of COL as well.

These intended model semantics carry over to our finite versions as well. We would like to emphasize that the precise semantics of the intended model are of not much consequence to our analysis; all we need are the basic encodings we gave in Section 2. In the sequel, by “finite language” we mean finite version of LDL or ELPS, or COL.

Definition 3.4. (Entailment). Given a program $P$ in the finite language and a database $D$, we say $(D, P) \models A$ iff the ground atom $A$ is in the intended model of $(P \cup D)$; here the database $D$ is viewed as a set of rules with null bodies.

Note that our encodings in Section 2 all conform to the intended models of the respective finite languages. The following lemma is therefore immediate, and we record it for future reference.

Lemma 3.1. (Set operations). There exist finite-language programs $P_1$ and $P_2$ such that for any database $D$:

- $(D, P_1) \models \text{UNION}(X, Y, Z)$ for set variables $X, Y$, and $Z$ iff $Z$ is the set union of $X$ and $Y$.
- $(D, P_2) \models \text{DU}(X, Y, Z)$ for set variables $X, Y$, and $Z$ iff $Z$ is the disjoint union of $X$ and $Y$.

Note that in the case of finite-ELPS, the encoding of the disjoint union $(DU)$ requires a negation in the built-in predicate “$=$” for disjoint set encoding (see Example 2.1). We show later that this built-in negation can be simulated by domain order and positive predicates only.

4. NONMONOTONICITY: GROUPING, DOMAIN ORDER + POSITIVE RULES

In this section we first show the power of grouping construct. Specifically, we show that the grouping construct can encode nonmonotonic queries such as EVEN and EQUAL that other languages (including nonmonotonic operators) such as stratified Datalog, bounded loops, etc., cannot simulate [6]. Next, we show an interesting interplay between positive rules with domain order, on the one hand, and nonmonotonic queries on the other. Specifically, we show that positive ELPS with domain order can simulate grouping of LDL. It is well known [22] that positive ELPS cannot simulate grouping; thus, adding domain order strictly increases the power of positive ELPS. The encodings used in this section also give a flavor of the (more complex) encodings in the coming sections. We record the definitions of the two problems here for future reference.

Definition 4.1. (EVEN and EQUAL). The problem EVEN takes a relation $R$ as an input and asks if the cardinality of $R$ is even. The problem EQUAL takes two relations, $R_1$ and $R_2$, as input and asks if $|R_1| = |R_2|$.
4.1. The Power of Grouping

Recall from Section 2 that grouping can encode disjoint union using positive predicates only. Another operation useful for our encodings is $Rem_1(X, X')$, which "removes" an element from $X$ to form $X'$. The element removed from $X$ is of no particular consequence, and hence $X'$ can be any of the possible sets that differ from $X$ in exactly one element. The following encodes $Rem_1$:

$$Rem_1(X, X') \leftarrow DU(X', \{x\}, X).$$

In the above, $DU$ stands for disjoint union. We show how naturally and easily grouping can express some well-known nonmonotonic queries such as EVEN and EQUAL (the definitions of these queries are given below). First note that grouping is a nonmonotonic operator in itself [9].

Observation 4.1. The grouping operation is nonmonotonic.

Proof. Consider a program consisting of a single rule $P(x, \{y\}^*) \leftarrow Q(x, y)$ and a database $D = \{Q(0, 1), Q(0, 2)\}$. Then $(D, P) \models Q(0, \{1, 2\})$. However, if we consider $D' = \{Q(0, 1), Q(0, 2), Q(0, 3)\}$, where $D' \supset D$, then clearly $(D' \cup P) \not\models Q(0, \{1, 2, 3\})$ and $(D' \cup P) \not\models Q(0, \{1, 2\})$. Hence, the program is not monotonic.

Since grouping is a nonmonotonic operation by itself, it may not be surprising that it encodes nonmonotonic queries such as EVEN or EQUAL; however, it is important to recall that several other nonmonotonic query languages such as (i) stratified-Datalog, (ii) while queries, and (iii) bounded-loop queries cannot express one or more of these queries [6]. This shows that grouping is a powerful and yet simple to understand nonmonotonic operator.

Theorem 4.2. (EVEN and EQUAL).

(i). There exists a program $P_{even}$ in finite-LDL such that for any relation $R$ as input, $(R \cup P_{even}) \models YES$ if $R$ has even cardinality; otherwise $(R \cup P_{even}) \not\models NO$.

(ii). There exists a program $P_{equal}$ in finite-LDL such that for any relations $R$ and $R'$ as input, $(R \cup R' \cup P_{equal}) \models YES$ if $|R| = |R'|$; otherwise $(R \cup R' \cup P_{equal}) \not\models NO$.

Proof. (i) Without loss of generality, assume that $R$ is monadic. Then the desired program $P_{even}$ is as follows:

$$\begin{align*}
Temp(1, x) & \leftarrow R(x) \\
Temp2(1, \{x\}^*) & \leftarrow Temp(1, x) \\
ODD([x]) & \leftarrow . \\
EVEN() & \leftarrow . \\
ODD(Z) & \leftarrow Rem_1(Z, Z'), EVEN(Z') \\
EVEN(Z) & \leftarrow Rem_1(Z, Z'), ODD(Z') \\
YES & \leftarrow Temp2(1, Z), EVEN(Z) \\
NO & \leftarrow Temp2(1, Z), ODD(Z)
\end{align*}$$

The first rule copies all the tuples of $R$ into a temporary relation $Temp$ such that the first attribute of every tuple of $Temp$ is 1. This is done to facilitate grouping of all the elements of $R$ in the second rule. Thus, $Temp2$ has only one tuple whose first component is 1 and whose second component is the set of all values of $R$. The next four rules encode $EVEN$ and $ODD$ in a recursive manner, using the predicate $Rem_1$ whose encoding we already presented. These rules are such that $EVEN(Z)$ holds for any set $Z$ if the cardinality of the set $Z$ is indeed even. $ODD(Z)$ is encoded analogously. The last two rules test if the set
obtained by grouping all the elements of $R$ is of cardinality even or odd.

Clearly, $(P_{\text{even}} \cup R) \models \text{YES}$ if $|R|$ is even and $(P_{\text{even}} \cup R) \not\models \text{NO}$ otherwise, as desired.

(ii) Again, without loss of generality, let both $R$ and $R'$ be monadic. We first encode a set of rules ($P_{\text{compare}}$) to compare the cardinalities of two sets, and then use this to obtain the desired encoding of $P_{\text{equal}}$:

\[
\begin{align*}
&EQ([],[]) \leftarrow \quad \{P_{\text{compare}}\} \\
&LESS([],X) \leftarrow x \in X \\
&EQ(X,Y) \leftarrow \text{Rem}_1(X,X'), \text{Rem}_1(Y,Y'), \\
&\quad EQ(X',Y') \\
&LESS(X,Y) \leftarrow \text{Rem}_1(X,X'), \text{Rem}_1(Y,Y'), \\
&\quad LESS(X',Y') \\
&\text{Temp}_1(1,x) \leftarrow R(x) \\
&\text{Temp}_2(1,x) \leftarrow R'(x) \\
&P(1,\{x\}^*) \leftarrow \text{Temp}_1(1,x) \\
&Q(1,\{x\}^*) \leftarrow \text{Temp}_2(1,x) \\
&\text{YES} \leftarrow P(1,X), Q(1,Y), EQ(X,Y) \\
&\text{NO} \leftarrow P(1,X), Q(1,Y), LESS(X,Y) \\
&\quad NO \leftarrow P(1,X), Q(1,Y), LESS(Y,X)
\end{align*}
\]

The rules $P_{\text{compare}}$ are such that for any pair of sets $X$ and $Y$, the predicate $LESS(X,Y)$ holds if $|X| < |Y|$, whereas $EQ(X,Y)$ holds if $|X| = |Y|$. We leave to the reader to verify that the program $P_{\text{equal}}$ is such that $(P_{\text{equal}} \cup R \cup R') \models \text{YES}$ if $|R| = |R'|$ and $(P_{\text{equal}} \cup R \cup R') \not\models \text{NO}$ otherwise. \hfill \qed

4.2. Finite-ELPS + Domain Order

It was shown in [22] that positive ELPS cannot simulate grouping. We show here that positive ELPS can simulate grouping if there is an order among the members to be grouped.

By an order on a set $S$ of $m$ elements, we mean the existence of predicates $\text{Next}$, $\text{Min}$, and $\text{Max}$ such that

\[
\text{Min}(a_1), \text{Next}(a_1,a_2), \ldots, \text{Next}(a_{m-1},a_m), \text{Max}(a_m)
\]

hold and $S = \{a_1, \ldots, a_m\}$.

The following theorem achieves the desired result.

**Theorem 4.3.** (Domain Order + Finite-ELPS simulates grouping). Let $D$ be any (monadic) relation on which an order is defined. There exists a positive program $P_{\text{group}}$ such that $(D \cup P_{\text{group}}) \models \text{Group}(X)$ iff $X$ is the set of all elements in $D$.

**Proof.** Since the elements of $D$ have an order, there exist predicates $\text{Min}$, $\text{Max}$, and $\text{Next}$ such that

\[
\text{Min}(a_1), \text{Next}(a_1,a_2), \ldots, \text{Next}(a_{m-1},a_m), \text{Max}(a_m)
\]

hold, where $D = \{a_1, \ldots, a_m\}$.

We prove the theorem in stages. We need two important primitives: (i) $nE(x,y)$, which holds iff $x \neq y$, and (ii) $nMem(x,X)$, which holds iff $x \notin X$.

The following rules encode $nE(x,y)$:
Note that in the above \( nE(x, y) \) holds between two elements \( x \) and \( y \) iff indeed \( x \neq y \). Once we have \( nE(x, y) \) defined only through positive rules, it is easy to see that \( DU \) ("disjoint union") can be defined using positive ELPS rules only. Next, we encode \( nMem(x, X) \), which holds iff \( x \notin X \).

\[
\begin{align*}
\text{P.notequal} & : nE(x, y) \leftarrow \text{Before}(x, y) \\
nE(x, y) & \leftarrow \text{Before}(y, x) \\
\text{Before}(x, y) & \leftarrow \text{Next}(x, y) \\
\text{Before}(x, y) & \leftarrow \text{Next}(x, z), \text{Before}(z, y)
\end{align*}
\]

We leave it to the reader to see that \( nMem(x, X) \) encodes \( x \notin X \).

To prove the theorem, the elements of the (monadic) relation \( D \) are to be grouped into a set such that \( Group(X) \) is true iff \( X = \{ x \mid x \in D \} \). This is done as follows: we generate all the subsets of \( D \), and then "recognize" the largest (by cardinality) element of them, which is nothing but the grouping of \( D \). In order to recognize the largest set, we need to be able to find the size of a set.

We define \( Size(X, k) \), which associates the cardinalities of sets to the ordered elements of \( Next \). Specifically, if \( |X| = k \), then, in the following program \( P.size \), the fact \( Size(X, y) \) will hold, where \( y \) is the \( k \)th element in the order defined by \( Next \):

\[
\begin{align*}
\text{P.size} & : \text{Size}(X, m) \leftarrow (X = \{x\}), \text{Min}(m) \\
\text{Size}(X, k) & \leftarrow \text{Rem}_1(X, X'), \text{Next}(k', k), \text{Size}(X', k')
\end{align*}
\]

Next, we obtain the members of the power set of \( D \) as elements of \( P \):

\[
\begin{align*}
P(\{\}) & \leftarrow \\
P(\{x\}) & \leftarrow D(x) \\
P(Z) & \leftarrow P(X), P(Y), \text{UNION}(X, Y, Z)
\end{align*}
\]

It is easy to see that the above rules compute the powerset of \( D \) into \( P \). Finally, we recognize the largest element of this power set and put it in \( Group \):

\[
\begin{align*}
\text{P.group} & : Group(X) \leftarrow P(X), \text{Size}(X, m), \text{Max}(m)
\end{align*}
\]

Clearly, the desired program \( P.group \) is given by the union of the fragments developed above. Specifically,

\[
P_{\text{group}} = (\text{P.notequal} \cup \text{P.nonMember} \cup \text{P.size} \cup \text{P.powerset} \cup \text{P.group}).
\]

This shows that grouping can be achieved on ordered sets using positive ELPS only.

The above theorem also gives a flavor of the techniques used to recognize various (more complex) maximums and minimums defined in the later proofs.

5. DATA COMPLEXITY

The notion of data complexity was introduced by Chandra and Harel [12] and was explored further by Vardi [28] and Immerman [18]. Informally, the data complexity of a query \( \Phi \) is the complexity of determining, for any given database \( D \), whether \( D \models \Phi \). This notion
extends naturally to programs as follows: given a program $PROG$ of a language $L$, the data complexity of $PROG$ measures the complexity of determining the answer of $PROG$ on any database.

In this section, we first define the notions of data complexity, exponential hierarchy, data completeness, etc., and then show that any $k$-level finite language is data complete for the $k$th level of the exponential hierarchy. The important part of this result is in establishing the lower bound, which in turn requires simulating the computations of a Turing machine. The latter requires total orders (or counters) of length arbitrarily high in the exponential hierarchy. In this section we assume that such orders are available a priori. The next section shows how to obtain such orders. Even though simulating Turing machine computations with orders available explicitly is intuitively clear, we provide these simulations in full detail to make the paper self-contained.

**Definition 5.1.** (Data complexity). Let $PROG \in L$ be a program in the language $L$. Let $YES$ be a special 0-ary predicate in $PROG$. Then the set

$$S_{PROG} = \{ D \mid (D, PROG) \models YES\}$$

is referred to as answer set of $PROG$. The data complexity of $PROG$ is the complexity of determining whether any given database $D$ belongs to $S_{PROG}$.

In order to specify the main result of the paper, we need two notions:

**Definition 5.2.** (Exponential hierarchy [15]). Let $\text{hyp}(c, n, 0) = n^c$ and $\text{hyp}(c, n, i + 1) = 2^{\text{hyp}(c, n, i)}$, for $i \geq 0$. The exponential hierarchy is the infinite sequence $E_0, \ldots, E_n, \ldots$, where

$$E_i = \bigcup_{c \in \mathbb{N}} \text{DTIME}(\text{hyp}(c, n, i))$$

Clearly, $E_0$ is $\text{PTIME}$ and $E_1$ is $\text{EXPTIME}$, etc.

**Definition 5.3.** (Data Completeness). A language $L$ is data complete for the complexity class $C$ if (i) for every $PROG \in L$, the data complexity of $PROG$ is in $C$ and (ii) there exists a program $PROG \in L$ such that the answer set of $PROG$ is a complete set for $C$.

The main result of the paper is that the $k$-level finite language (i.e., finite-ELPS, finite-LDL, or COL) is data complete for the $k$th level of the exponential hierarchy.

This result is shown by first showing that the data complexity of every $k$-level program is in $E_k$, and then exhibiting a $k$-level program $PROG$ whose answer set is a complete set for $E_k$. The former establishes the containment in $E_k$ and the latter establishes the $E_k$-hardness. We prove the containment result by obtaining a bound on the maximum size of the Herbrand universe of any program; this turns out to be straightforward.

The difficult part is to prove the hardness result. This is proved as follows. Let $M$ be any Turing machine that runs in $E_k$. Then we show that there exists a $k$-level program $PROG$ such that for a fixed 0-ary predicate $YES$ the following holds:

for any $\bar{x}$ there exists a database $D_\bar{x}$ such that

$$(D_\bar{x}, PROG) \models YES \text{ iff } M \text{ halts on } \bar{x}.$$  \hspace{1cm} (5.1)

Interestingly, the program $PROG$ does not need negation, even on base or built-in
predicates. Clearly, by choosing $M$ to be such that its language is a complete set for $E_k$, we achieve the desired result, i.e., exhibiting a $k$-level generating program whose answer set is complete for $E_k$. A simulation that achieves the equation (5.1) will be discussed in the following section.

5.1. Data Complexity—Turing Machine Simulations

In this section we prove the main result that the $k$-level generating language is data complete for the $k$th level of the exponential hierarchy. The following theorem records this result.

Theorem 5.1. (Data complexity of the $k$-level finite language). The $k$-level finite language is data complete for the $k$th level of the exponential hierarchy.

The upper bound (i.e., containment in $E_k$) is straightforward. For any $k$-level program, let $HERB(n)$ denote the size of the Herbrand universe of $(PROG \cup D)$, where $D$ is the input database of size $n$. Then the set of all the facts that can ever be entailed by the program is a polynomial in $HERB(n)$. This is because, the program has a fixed number of relations with some fixed (maximum) arity. Furthermore, it is easy to verify that $|HERB(n)|$ is a $k$-level exponent in $n$. Hence, a naive bottom-up generation of all the facts entailed by the program and testing for the desired fact can be carried out in deterministic time determined by $E_k$. Thus, the upper bound of the $k$-level language is established.

To prove Theorem 5.1 we therefore need to establish the more interesting part, namely, the hardness result. This is established in Lemma 5.2 below by proving equation (5.1). It will be clear from our construction that $D_2$ in (5.1) can be computed in polynomial time in the size of the input $\tilde{s}$. By choosing $M$ in (5.1) to be the one whose language is a complete set for $E_k$, we obtain the desired result. As mentioned earlier, in this section we show how to carry out Turing machine simulations assuming that an order is given. The next section shows how to obtain the requisite orders. Intuitively, orders of sufficient length are obtained by repeatedly applying a procedure that takes an order of length $n$ (that is encoded in $D_2$ of (5.1) above) and produces another of length $2^n$. Theorem 6.1 shows that it is indeed possible to achieve this using only positive rules. The following lemma completes the proof of Theorem 5.1.

Lemma 5.2. For any Turing machine $M$ that runs in $E_k$, there exists a program $PROG$ in the $k$-level finite language such that

for any $\tilde{s}$ there exists a database $D_2$ such that

$$(D_2, PROG) \models YES \text{ iff } M \text{ halts on } \tilde{s}. \quad (5.1)$$

Proof. As stated earlier, we assume that the requisite total order is available; the next section shows how to obtain such order. The desired program $PROG$ has some special predicates that simulate the tape contents and the control head of $M$.

- $TAPE(s, p, t)$ is true iff the tape of $M$ has symbol $s$ in position $p$ at time $t$.
- $HEAD(q, p, t)$ is true iff the control head of $M$ has symbol $s$ in position $p$ at time $t$.
- $\delta(q, s, q', s', \Delta)$ is the transfer function of $M$, i.e., if the state of the control head is $q$ and the tape symbol under the head is $s$, then the machine writes $s'$ and moves $\Delta$ steps. ($\Delta$ is $+1$ or $0$ or $-1$).

Assume that $M$ is in $DTIME(f(n))$, where $n$ is the size of any input. Thus on $\tilde{s}$ it runs for at most $f(|\tilde{s}|)$ time.
Let $\text{NEXT}(x, y)$ denote the counter of length $f(\bar{s})$, i.e., there exists a sequence (for $N = f(\bar{s})$)

\[
\text{\textit{MINIMUM}}(1), \text{\textit{NEXT}}(1, 2), \ldots, \text{\textit{NEXT}}(N - 1, N), \text{\textit{MAXIMUM}}(N)
\]
of length $N$ that can be used to simulate the Turing machine $M$. Here $\text{\textit{MINIMUM}}(.)$ and $\text{\textit{MAXIMUM}}(.)$ are used to indicate the first and the last elements of the order (i.e., the end points). The order is expressed in terms of natural numbers for convenience, whereas the actual order consists of nested sets of the members of the input $\bar{s}$. Thus for example, if $\bar{s}$ is $(1, 2)$, then the order NEXT is of the form:

\[
\text{\textit{MINIMUM}}([]), \text{\textit{NEXT}}([], \{1\}), \text{\textit{NEXT}}([1], \{2\}), \text{\textit{NEXT}}([2], \{1, 2\}), \text{\textit{MAXIMUM}}([1, 2])
\]

The sequence $\text{\textit{NEXT}}(\cdot, \cdot)$ is not available to us to begin with. It is generated from an initial sequence

\[
N(s_1, s_2), \ldots, N(s_{n-1}, s_n)
\]
on the members of the input $\bar{s} = s_1, \ldots, s_n$. This initial sequence is a part of the database $D$ as shown below. The next section (Theorem 6.1) describes how to obtain the order of length $f(n)$ from this initial order of length $n$. This extension is crucial to our simulation. For now we assume that the order $\text{\textit{NEXT}}$ of the desired length is available to us. We use another predicate $\text{\textit{PREV}}$, defined as below, for convenience:

\[
\text{\textit{PREV}}(x, y) \leftarrow \text{\textit{NEXT}}(y, x)
\]

<table>
<thead>
<tr>
<th>Initial Tape and Head Configuration</th>
<th>Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{\textit{HEAD}}(q_i, 1, 1)$</td>
<td>$\text{\textit{Min}}(s_1)$</td>
</tr>
<tr>
<td>$\text{\textit{TAPE}}(s_1, 1, 1)$</td>
<td>$\text{\textit{N}}(s_1, s_2)$</td>
</tr>
<tr>
<td>.</td>
<td>$\ldots$</td>
</tr>
<tr>
<td>.</td>
<td>$\text{\textit{N}}(s_{n-1}, s_n)$</td>
</tr>
<tr>
<td>$\text{\textit{TAPE}}(s_m, m, 1)$</td>
<td>$\text{\textit{Max}}(s_n)$</td>
</tr>
</tbody>
</table>

The database $D$ also encodes the initial tape contents of $M$ into the predicate $\text{\textit{TAPE}}$. Note that $q_i$ is the initial state of $M$. Thus, $\text{\textit{HEAD}}(q_i, 1, 1)$ indicates that the control head is in the initial state $q_i$ at time 1 and is “looking” at the position 1 of the tape. Similarly, $\text{\textit{TAPE}}(s_i, i, 1)$ indicates that the TAPE of $M$ has symbol $s_i$ is in position $i$ at the initial time 1.

The following rules, constituting the program $\text{\textit{PROG}}$, simulate the computations of the Turing machine $M$. We write “$\textit{head}_1, \textit{head}_2 \leftarrow \textit{body}”$ to abbreviate the two rules “$\textit{head}_1 \leftarrow \textit{body}”$ and “$\textit{head}_2 \leftarrow \textit{body}.” We use the same counter $\text{\textit{NEXT}}$ to count both the tape cells and the time. Without loss of generality we assume that there is only one final accepting state, denoted by $q_A$. 
YES ← HEAD(q_a, p, t),

HEAD(q', p', t'), TAPE(s', p', t') ← HEAD(q, p, t),
   TAPE(s, p, t), δ(q, s, q', s', +1), NEXT(t, t'), NEXT(p, p'),  
   \tag{5.2}

HEAD(q', p', t'), TAPE(s', p', t') ← HEAD(q, p, t),
   TAPE(s, p, t), δ(q, s, q', s', -1), NEXT(t, t'), PREV(p, p'),  
   \tag{5.3}

HEAD(q', p', t'), TAPE(s', p', t') ← HEAD(q, p, t), TAPE(s, p, t),
   δ(q, s, q', s', 0), NEXT(t, t'),  
   \tag{5.4}

TAPE(s, p, t') ← HEAD(q, p', t), TAPE(s, p, t), NEQ(p, p'),
   NEXT(t, t'),  
   \tag{5.5}

TAPE(blank, x, 1) ← Greater(x, m), Max(m),
   \tag{5.6}

δ(q, s, q', s', Δ) ← .  
   \tag{5.7}

In the above rules, δ is the transfer function, i.e., δ(q, s, q', s', Δ) indicates that if the control head of M is in the state q and looking at the symbol s on the tape, then it writes s' on the tape and moves Δ steps forward (Δ can be +1 or -1 or 0). The transfer function constitutes a part of the program PROG as indicated above. Each member of the transfer function constitutes a rule (with empty body) in the program PROG. Thus, rule (5.8) in PROG is actually a set of rules (with empty bodies) each corresponding to a transition in the transition function δ of M.

Rule (5.2) succeeds when the machine reaches the final accepting state. Rules (5.3), (5.4), and (5.5) simulate the computations of the Turing machine for the three different values of Δ. Rule (5.6) copies over the information that remains unchanged, i.e., the cells that are not directly below the head remain unchanged. Rule (5.7) puts "blanks" in those cells of TAPE that are not filled with s. The predicate Greater(x, m) essentially tests if x is more than the last symbol of s. Note that Greater(., .) can be easily encoded using the order. Thus every other tape cell gets blank.

The only new predicate that we introduced is NEQ(x, y), which tests if x ≠ y. Please recall that we showed how to encode this predicate using NEXT in Section 4.2. Thus, PROG is such that

for any s there exists a database D_s such that

(D_s, PROG) ⊨ YES iff M halts on s.  
   \tag{5.1}

This verifies this lemma, and from the discussion preceding the lemma, it also establishes Theorem 5.1. □

6. EXTENDING ORDERS

Note that Theorem 5.1 assuming an order of sufficient length is available. This section shows how to obtain such an order. Note that an order of length n (the size of the input) can always be encoded into D_f (of equation (5.1) in the previous section). Hence, the crucial step, established in this section, is to extend an order of length n to an order of length 2^n using only positive rules with 1-level predicates. Because this process can be repeated as
many times as needed, a $k$-level program can obtain an extended order that can simulate any Turing machine in $E_k$ by only starting with an initial order of length equal to the size of the input.

Our approach to obtain these orders differs from the previous ones [10, 16, 24] in the literature that involved EXPTIME-hardness results: although the Hypothetical Datalog of Bonner [10] involves results about EXPTIME-Hardness, the simulations involved only PSPACE (alternating) Turing machines; hence, no explicit extension of an order beyond the order among the domain constants was required. The approaches of Kuper and Vardi [24] and of Hull and Su [16] use negation and "B-" to extend an order of length $n$ to another of length $2n$. However, we do not have access either to negation or to "B-", and hence employ different techniques to extend orders.

We use set nesting to extend an order of length $n$ to another of length $2^n$. By an order of length $n$ we mean a binary predicate giving the successive elements of the order, and two other predicates, $\text{MIN}(x)$ and $\text{MAX}(x)$, giving the end points of the order. When we say that "a program generates/extends an order of length $n,"$ we mean that the program entails predicates that define an order and its end points.

The basic idea of this extension is as follows. Let $N(x, y)$ denote the successive elements in the original order among the elements of $D$, the domain of the input database. Also, let $\text{Min}$ and $\text{Max}$ denote the minimum and maximum values of $D$ respectively (i.e., they denote the end points of $N$). Let the successive elements in the desired new order of length $2^{|D|}$ be denoted by $\text{NEXT}(x, y)$, and let the end points of this new order be denoted by $\text{MAXIMUM}$ and $\text{MINIMUM}$. This new order is obtained by extending the original order $N$ to the members of the power set $2^D$ of $D$. Figure 1 illustrates the power set and the new order induced on it by the original order for $D = \{1, 2, 3, 4\}$. The new order is defined from left to right in each "layer" and from top to bottom across layers. Thus, $\text{NEXT}([2, 1], [3, 1])$ is true and, similarly, $\text{NEXT}([4, 3], [321])$ is true, etc.

**Definition 6.1.** (The new total order). To each member $s$ of the power set, associate a number in radix $|D|$ obtained by sorting the elements of $s$ in decreasing order. Then the natural order obtained on these associated numbers is the desired order on the elements of the power set of $D$.

Clearly, each $X \in 2^D$ gets a unique natural number in the radix $|D|$. Thus, the set $\{1, 2\}$ is interpreted as 21 (i.e, twenty-one). Similarly, the set $\{1, 2, 3\}$ is interpreted as 321. Clearly, each set gets a unique number. The empty set gets the number zero, and the largest set (that is, the set containing every element of $D$) gets the highest number.

**End Points of the Order.** The end points of this order are denoted by the predicates $\text{MAXIMUM}$ and $\text{MINIMUM}$ and are obtained as follows. The $\text{MINIMUM}$ is ob-
tained by simply asserting that

\[ MINIMUM(\{\}) \leftarrow . \]

The maximum requires grouping all the members of \( D \) into an element. This is done by following the same procedure as done in Theorem 4.3: essentially, first generate the entire power set of \( D \) and then "recognize" the largest element using a "Size" predicate. Please refer to Section 4 for details.

In the sequel, any set (denoted by a capital letter variable) is assumed to be a member of the power set of \( D \). There are certain important properties that the new order on these sets satisfies. These properties, given below, play a crucial role in obtaining it.

Maximum Digit. A member \( s \in 2^D \) can be viewed both as a set and as a number in radix \( |D| \). However there is a difference between these numbers and the "regular" numbers: no digit appears repeatedly in the numbers represented by members of \( 2^D \). This is because, these "numbers" are in fact sets. Thus, each set/number has a single "maximum" or "largest" digit. We can obtain the maximum of any number/set using positive rules only. Because this operation, denoted by \( MaxOf(X, x) \), is useful later, we present the program that obtains it.

\[
\begin{align*}
MaxOf(\{x\}, x) & \leftarrow . \\
MaxOf(X, x) & \leftarrow MaxUnder(X, x, m), Max(m) \\
MaxUnder(X, k, k) & \leftarrow k \in X \\
MaxUnder(X, x, k) & \leftarrow nMem(k, X), MaxUnder(X, x, k'), N(k', k).
\end{align*}
\]

Rule (6.1) is obvious. Rules (6.3) and (6.4) define a predicate "\( MaxUnder(X, x, k) \)" which has the following semantics: (i) \( x \leq k \); (ii) for every \( y \leq k \) and \( y > x \), it is the case that \( y \notin X \). Thus, \( MaxUnder(X, x, k) \) is true iff \( x \) is the largest among those elements of \( X \) that are less than or equal to \( k \). Rule (6.3) trivially conforms to this semantics, whereas rule (6.4) enforces this semantics through recursion. It uses a predicate \( nMem(k, X) \) to capture "\( k \notin X \)." Recall that \( nMem \) can be defined using positive rules only, as described in Section 4.2.

Clearly, then \( x \) is the largest element of \( X \) if \( MaxUnder(X, x, m) \) is true, where \( m \) is the largest element that can be present in any set \( X \). Rule (6.2) encodes \( MaxOf(X, x) \) using exactly this observation.

Largest and Smallest Numbers/Sets. The largest \( k \)-digit number of \( 2^D \) is referred to as layer maximum because it is the maximum of the \( k \)th layer (counted from the top). In our example, \{4, 3, 2\} is the layer maximum for the third layer, whereas \{4, 3\} is the layer maximum for the second layer (see Figure 1). It is easy to see that a number/set is a layer maximum iff it forms a descending sequence of numbers from \( m \), the maximum of \( D \). This is unlike the "regular" numbers where the largest \( k \)-digit number is obtained by repeating \( m \), the maximum of \( D \), \( k \) times.

A \( k \)-digit number \( X \) is called local maximum if the following holds: let \( x \) be the largest element of \( X \) and let \( S_x \) be the set of all \( k \)-digit numbers/sets whose largest element is \( x \). Then \( X \) is the largest number of all numbers/sets in \( S_x \). Thus, in our example, \{4, 3\} is a local maximum. This is because, the two-digit numbers/sets with 4 as the maximum digit are \{4, 1\}, \{4, 2\}, and \{4, 3\}. Clearly, \{4, 3\} is the largest of them (it represents forty-three, whereas the remaining two represent forty-one and forty-two). Hence it is a local maximum. Similarly, \{3, 2\} is a local maximum. Note that \{3, 2\} is not a layer maximum. Essentially, a local maximum forms a decreasing sequence starting with its maximum digit, whereas a
layer maximum forms a decreasing sequence starting with \( m \), the maximum element of \( D \). Layer maximums and local maximums play an important role in extending an order. The following program obtains layer maximums and local maximums in a straightforward way by testing for decreasing sequences (DecSequence).

\[
\begin{align*}
\text{LayerMax}(X) & \leftarrow \text{DecSequence}(X, m), \text{Max}(m) \\
\text{LocalMax}(X) & \leftarrow \text{MaxOf}(X, x), \text{DecSequence}(X, x) \\
\text{DecSequence}(X, k) & \leftarrow \text{DU}(X', \{k\}, X), \text{Next}(k', k), \text{DecSequence}(X', k') \\
\text{DecSequence}([], k) & \leftarrow .
\end{align*}
\]

A layer minimum and local minimum can be defined analogously. A \( k \)-digit number is a layer minimum if it is the smallest of all the \( k \)-digit numbers. Thus it forms an increasing sequence of numbers from \( m \), the minimum of \( D \). A \( k \)-digit number \( X \) is local minimum if the following holds: if \( x \) is the maximum digit of \( X \), then \( X \) is the smallest number of all \( k \)-digit numbers that contain \( x \) as the maximum number. These minimums can be encoded analogous to the two kinds of maximums above and we leave the details to the reader.

The following theorem shows that using an order of length \(|D|\), we can obtain a new order of length \( 2\|D\| \).

**Theorem 6.1.** (Extending an order). Let \( D \) be a set of constants given along with a total order on its members. Then there exists a \( 1 \)-level testing program with only positive rules that extends this order to a length \( 2\|D\| \).

**Proof.** Let \( N(x, y) \) denote the order among the elements of \( D \). Also, let \( \text{Min}(.) \) and \( \text{Max}(.) \) denote the minimum and maximum values of \( D \), respectively. The desired order of length \( 2\|D\| \) is denoted by \( \text{NEXT}(x, y) \). This new order is obtained by extending the order on \( D \) given by \( N \) to an order on \( 2^D \) using Definition 6.1. This new order is represented by \( \text{NEXT} \).

We showed earlier how to compute the end points of this new order. Now we show how the other members of the power set are put in the new order \( \text{NEXT} \). Constructing the new order between two numbers/sets of \( 2^D \) can be thought of as putting a directed edge between those members.

We first put the edges between the singleton sets (e.g., \( \text{NEXT}([2], [3]) \)):

\[
\text{NEXT}([x], [y]) \leftarrow N(x, y).
\]

This rule extends the order to the elements in the first layer (see Figure 1).

However, rule (6.5) does not extend the order to the elements in the second layer. The following rule extends the order to some elements of the second layer. This rule takes numbers/sets \( X' \) and \( Y' \) such that \( \text{NEXT}(X', Y') \) is true, and forms \( X \) and \( Y \) from these two, respectively, by adding an element \( x \) that is larger than any element in \( X' \) or \( Y' \), and concludes that \( \text{NEXT}(X, Y) \) is true:

\[
\begin{align*}
\text{NEXT}(X, Y) & \leftarrow \text{DU}(X', \{x\}, X), \text{DU}(Y', \{x\}, Y), \text{MaxOf}(X, x), \text{MaxOf}(Y, x), \\
& \quad \text{NEXT}(X', Y')
\end{align*}
\]

It is clear from the definition of order that rule (6.6) produces \( \text{NEXT}(X, Y) \) correctly (i.e., if \( \text{NEXT}(X, Y) \) is produced, then indeed \( Y \) is adjacent to \( X \) according to Definition 6.1). Furthermore, rule (6.6) extends order to some of the elements in the second layer. For example, since \( \text{NEXT}([1], [2]) \) is generated by rule (6.5) and since 3 is the maximum of both \( [1, 3] \) and \( [3, 2] \), it follows from rule (6.6) that \( \text{NEXT}([3, 1], [3, 2]) \) must hold. Similarly, \( \text{NEXT}([4, 1], [4, 2]) \) and \( \text{NEXT}([4, 2], [4, 3]) \) are produced due to rule (6.6).
Specifically, if $S_x$ denotes the set of numbers/sets of the second layer that have $x$ as their maximum digit, then (6.6) extends the new order to every number/set in $S_x$. This is proved as follows:

Consider any two numbers/sets $X$ and $Y$ in $S_x$ such that they should be adjacent in the new order. Then, obtain $X'$ and $Y'$, respectively, from $X$ and $Y$ by deleting $x$ from them. Clearly, from Definition 6.1, it follows that $X'$ and $Y'$ must be adjacent to each other. However, because $X'$ and $Y'$ are singleton sets, from rule (6.5) it follows that indeed $\text{NEXT}(X', Y')$ is already generated. Hence, $X$ and $Y$ satisfy the body of rule (6.6) and hence indeed $\text{NEXT}(X, Y)$ is generated.

However, rule (6.6) does not extend the order to $X$ and $Y$ if they are not in the same $S_x$. For example, although $\text{NEXT}([3, 2], [4, 1])$ should be true, it is not produced by rule (6.6). Thus, to extend the order to all elements in the second layer, we need to connect the maximum of $S_x$ to the minimum of the adjacent $S_y$. Clearly, the maximum of $S_x$ is a local maximum, and the minimum of $S_y$ is a local minimum. The rule (6.7) connects $X$ and $Y$ if $X$ is a local maximum and $Y$ is an adjacent local minimum. Because this relation holds for any pair of adjacent local maximum and local minimum, we use rule (6.7) to connect every pair of adjacent local maximum and local minimum (without restricting to only layer 2):

$$\text{NEXT}(X, Y) \leftarrow \text{LocalMax}(X), \text{LocalMin}(Y), \text{Eqsize}(X, Y), \text{MaxOf}(X, x), \text{MaxOf}(Y, y), N(x, y)$$

(6.7)

In the above, $\text{Eqsize}(X, Y)$ makes sure that $X$ and $Y$ are in the same layer. The following rules encode $\text{Eqsize}(X, Y)$:

$$\text{Eqsize}([], []) \leftarrow .$$
$$\text{Eqsize}(X, Y) \leftarrow \text{DU}(X', [x], X), \text{DU}(Y', [y], Y), \text{Eqsize}(X', Y').$$

Recall that $\text{DU}(X, Y, Z)$ stands for the fact that $Z$ is the disjoint union of $X$ and $Y$.

We show that rules (6.6) and (6.7) extend the order fully, not only to the second layer, but to every other layer as well. Formally, we prove that if $X$ and $Y$ are two numbers/sets belonging to the same layer, and are adjacent according the new order, then indeed rules (6.6) and (6.7) together produce $\text{NEXT}(X, Y)$. We prove this by induction on the number of layers.

Certainly, it holds for the first layer. Suppose, as an induction hypothesis, (6.6) and (6.7) extend the new order fully to the $i$th layer. Then consider the numbers/sets of $(i + 1)$th layer. Let $X$ and $Y$ be two numbers/sets in $(i + 1)$th layer such that they should be adjacent according to the definition of order (Definition 6.1). There are two cases: either $X$ and $Y$ have the same maximum value, or they do not. In either case, we show that indeed $\text{NEXT}(X, Y)$ is produced, thus showing that the rules (6.6) and (6.7) extend the order to $(i + 1)$th layer.

Consider the case when $X$ and $Y$ have the same maximum value, where $x$ denotes this maximum value. Thus, $X$ and $Y$ are in $S_x$, where $S_x$ is the set of all elements of $(i + 1)$th layer with $x$ as their maximum element. We show that if $X$ and $Y$ are next to each other according to the order defined earlier (Definition 6.1), then the rules (6.6) and (6.7) indeed produce $\text{NEXT}(X, Y)$. This is as follows. First obtain $X'$ and $Y'$ by deleting $x$ from them. By definition of the order (Definition 6.1), $X'$ and $Y'$ should be adjacent to each other. Because $X'$ and $Y'$ belong to the $i$th row, by induction hypothesis, it follows that $\text{NEXT}(X', Y')$ is already produced. Hence, from rule (6.6) it follows that $\text{NEXT}(X, Y)$ holds. Thus, the new order extends to the elements within $S_x$ for any $x$.

Now consider $X$ and $Y$ that do not have the same maximum value. Clearly, then $X$ and
Y will be next to each other in the new order iff X is the maximum of all the numbers in $S_x$ and Y is the minimum of all the numbers in $S_y$ and furthermore, x and y are adjacent. Then, clearly, from the rule (6.7), $\text{NEXT}(X, Y)$ holds.

Thus, for any X and Y in the $(i + 1)$th layer, if X and Y should be adjacent in the new order, then indeed $\text{NEXT}(X, Y)$ holds.

Although (6.5), (6.6), and (6.7) above extend the order fully to each layer, they do not extend the order across layers. For example, $\{4, 3\}$ is not connected to $\{3, 2, 1\}$. Thus, if X is the maximum element of one layer and Y is the minimum element of the next layer, then clearly, $\text{NEXT}(X, Y)$ must be true for X and Y. The following rule achieves this:

$$\text{NEXT}(X, Y) \leftarrow \text{LayerMax}(X), \text{LayerMin}(Y), \text{LESS1}(X, Y) \quad (6.8).$$

In the above, $\text{LESS1}(X, Y)$ is true iff the size of Y is exactly one more than the size of X. This makes sure that Y is indeed in the next layer. The predicate $\text{LESS1}(X, Y)$ can be easily encoded using positive rules:

$$\text{LESS1}([x], []) \leftarrow . \quad \text{LESS1}(X, Y) \leftarrow \text{DU}(X', [x], X), \text{DU}(Y', [y], Y), \text{LESS1}(X', Y').$$

From (6.5-6.8) it follows that $\text{NEXT}(., .)$ indeed defines an order of length $2^{[D]}$. Note that all the predicates used in rules (6.5-6.8) have already been defined using only positive rules and grouping. □

7. CONCLUSIONS

In this paper we obtained both the upper and lower bounds of the data complexity of three rule-based logic programming languages that are proposed in the literature for reasoning with databases containing nested sets and tuples. The three languages we consider are ELPS, LDL, and COL. We define a notion of "k-level" language and show that the data complexity of k-level COL, k-level LDL, and k-level ELPS is complete for the $k$th level of the exponential hierarchy. There are two aspects to our results:

First, they are different from what one would expect from similar such results—from the fact [14, 16] that higher-order calculi (such as those of [16]) can simulate fixed points, one would expect (as in [5]) that our k-level programs would have the same data complexity as $\text{CALC}_k$ of [16]. However, it turns out not to be the case; for instance, $\text{CALC}_k$ does not have tight upper and lower bounds (see [16] for details), whereas we exhibit tight bounds for data complexity of our k-level languages.

Second, the techniques we use are different from those used for similar results in the literature (such as [10, 16, 23]). Furthermore, the techniques we use yield the following interesting "by-products": (i) The power of grouping: We show that the finite-LDL, with the help of its grouping rule, can express nonmonotonic queries such as EQUAL and PARITY. Contrast this with the fact [6] that at least one of PARITY or EQUAL cannot be expressed by a number of nonmonotonic query languages such as stratified-Datalog, fix-point queries, bounded-loop queries, etc. (ii) The interplay between positive rules with domain order, and nonmonotonicity: We show that with domain order even positive rules encode a number of nonmonotonic operations.
The author would like to acknowledge the constant encouragement and support of Tomasz Imieliński and Shamim Naqvi. Thanks also go to the reviewers whose comments helped to clarify parts of the paper. The author would like to acknowledge the NSF grants IRI93-09320 and IRI93-342-2109 that provided partial support to him.

REFERENCES