SET CONSTRUCTORS IN A LOGIC DATABASE LANGUAGE*

CATRIEL BEERI,† SHAMIM NAQVI,
ODED SHMUELI,‡ AND SHALOM TSUR

Logical Data Language—is a language developed at MCC. The language supports various extensions of Horn-clause-based programming. In particular, it allows the use of negation, and the use of sets—either in enumerated form or as a result of element grouping. This paper investigates how semantics can be defined for the language. More broadly, it examines the intricacies arising from the introduction of set constructs into a logic-based language. The concept of a model is extended to account for the set constructs. It is shown that a program may have no model, or it may have several incomparable models. A syntactic restriction on programs, called layering, is introduced, and it is shown that, for programs satisfying this restriction that have models, there exists a minimal model that is, in a well-defined sense, preferable to all other models of the program. This model can be constructed bottom-up. Next, conditions guaranteeing that a program has models are presented. Finally, relationships to other language proposals, and relative merits of language constructs, are briefly considered.

1. INTRODUCTION

The logic-programming paradigm and its use in data- and knowledge-base systems have been the focus of intense research and development efforts in recent years.

†Work partially supported by a grant from the United States–Israel Binational Science Foundation (BSF), Jerusalem, Israel.
‡This research was partially supported by the fund for the promotion of research at the Technion.

Address correspondence to Catriel Beeri, Department of Computer Science, The Hebrew University, Jerusalem, Israel.
Accepted July 1990.

THE JOURNAL OF LOGIC PROGRAMMING
655 Avenue of the Americas. New York, NY 10010

(0743-1066/91/$3.50)
These efforts deal with the following:

(1) removing certain nonlogical features of classical languages like PROLOG;
(2) developing global optimization strategies for programs that manipulate large amounts of data;
(3) providing well-defined semantics for programs that contain features that are not allowed in Horn-clause programs, such as negation or set elements and set operators.

This paper deals with the last subject. Specifically, we describe the problems that arise when sets and set operators are introduced into logic programming, and present solutions.

The results reported here were obtained as part of the work on _LDL_ (Logical Data Language), a language that has been under development at MCC since 1985 and has undergone two implementations. _LDL_ is an attempt to combine the benefits of logic programming with those of relational query languages. For a description of the language, including motivation and explanation of the various features, see [36, 27]. We introduce below the use of sets and set operators in logic programming by means of some examples, written in _LDL_. However, _LDL_ is not the subject of this paper. The paper concentrates on a formal treatment of the difficulties that arise in assigning meaning to programs that use certain set constructs, and provides a unified framework in which the semantics of Horn clauses with sets and negation can be described. The results are applicable to any language that combines sets and logic programming.

We introduce the essential features of general _LDL_ programs, and some of the associated problems, via examples. The starting point is a language that has come to be called Datalog. It is similar to PROLOG, but it does not contain the extralogical features of that language, e.g., cut. Also, unlike PROLOG, the programmer does not have explicit control over the order of execution of the literals within a rule or the order of execution of the rules. Hence, Datalog programs have a pure declarative semantics [4]. The _ancestor_ example is by now a classical program of this language:

\[
\text{ancestor}(X, Y) \leftarrow \text{parent}(X, Y).
\]
\[
\text{ancestor}(X, Y) \leftarrow \text{ancestor}(X, Z), \text{parent}(Z, Y).
\]

To augment the expressive power of the language, we allow function symbols, and negated literals in the body of rules. The following is a program that derives an "exclusive ancestor" relation, i.e., all ancestors except those that are also ancestors of a particular individual (the binding to \(Z\)):

\[
\text{ancestor}(X, Y) \leftarrow \text{parent}(X, Y).
\]
\[
\text{ancestor}(X, Y) \leftarrow \text{parent}(X, Z), \text{ancestor}(Z, Y).
\]
\[
\text{excl} \_\text{ancestor}(X, Y, Z) \leftarrow \text{ancestor}(X, Y), \neg \text{ancestor}(X, Z).
\]

This program satisfies the restriction of layering, or stratification [2, 14, 42, 26]. It can be viewed as consisting of two layers: the first layer contains the first two _ancestor_ rules and the _parent_ base relation, and the second layer contains the _excl_ _ancestor_ rule. Negation is applied to a predicate of the first layer, only to compute a predicate of the second layer, so the second layer may be unambiguously computed once the first one is fully computed. Thus, the program can be assigned a "natural" iterated fixpoint semantics, as described in those papers.
However, it is not always possible to associate such a semantics with a program that uses negation, as illustrated by the following program \([s(X)\) denotes the successor of \(X\)]:

\[
\begin{align*}
\text{int}(0). \\
\text{int}(s(X)) & \leftarrow \text{int}(X). \\
\text{even}(0). \\
\text{even}(s(X)) & \leftarrow \text{int}(X), \neg \text{even}(X).
\end{align*}
\]

In this program, the predicate \(\text{even}\) depends on a negative occurrence of itself, which implies that there is no partition of the rules (and predicates) of the program into layers, in the sense above.

Sets and set manipulation occur naturally in many applications. In particular, query languages and data manipulation in database systems are set oriented. In classical logic-programming languages, sets can be represented only indirectly, by lists. This representation is awkward, and does not support optimization techniques such as those developed for Datalog. A primary goal of [LDL] was to introduce sets and set operators as first-class citizens of the language. Sets can appear as elements of base relations, as proposed in nested-relations and complex-object models [1, 6, 16]. Predicates such as membership, union, and so on, can be used on such elements. Sets as arguments of predicates can also be generated by using appropriate constructors. The Boolean set operations are, of course, such constructors. In [LDL] two additional constructors are used: set enumeration and set grouping. Set enumeration is the process of constructing a set by enumerating its elements, as in the following example:

**Example.** The relation \(\text{book}\) includes tuples whose first component is the title of a book, and whose second component is the price of the book. The relation \(\text{book\_deal}\) is composed of tuples with a single (set) component. This set component includes the titles of three books whose total price is less than $100. The [LDL] program defining the relation \(\text{book\_deal}\) is shown below:

\[
\begin{align*}
\text{book\_deal}((X, Y, Z)) & \leftarrow \text{book}(X, P_x), \\
& \quad \text{book}(Y, P_y), \\
& \quad \text{book}(Z, P_z), \\
& \quad X \neq Y, X \neq Z, Y \neq Z, \\
& \quad P_x + P_y + P_z < 100.
\end{align*}
\]

Grouping is a common and useful operation in databases. It is used to partition the tuples in a relation into sets, grouped according to common values in some columns. For example, given a parts-and-suppliers relation, we might want to group it by common supplier number, so as to have for each supplier the set of all parts he supplies. In a model that allows sets as elements, the result of such a grouping operation is a (nested) relation, in which the value in the first column of a tuple is a supplier number, and the value in the second column is a set of part numbers. The following program expresses this query in [LDL]:

\[
\begin{align*}
\text{part\_sets}(S\#, (P\#)) & \leftarrow \text{supplier}(S\#, P\#).
\end{align*}
\]

The \((\cdots)\) in the head is the grouping operator. The intended meaning is to find, for each \(S\#\), all substitutions that satisfy the body, to collect the \(P\#\)-values in
them into a set, and to construct a tuple in the result from each $S#$ and the corresponding set of $P#$'s.

Unlike set enumeration, where the set is constructed by listing its elements, in grouping the set is constructed by defining its elements by a property (i.e., a conjunction of predicates) that they satisfy. Thus, set grouping follows the standard mathematical style of defining sets by the properties of their elements. It follows that the cardinality of the grouped set in a derived relation is unbounded (and may be infinite, since we allow function symbols). Intuitively, grouping seems to be more powerful than the other set operators we have mentioned.

To gain familiarity with the grouping construct, consider the following simple example:

\[ p(a, b). \]
\[ p(a, c). \]
\[ q(X, \langle A \rangle) \leftarrow p(X, A). \]

Then the only tuple in the relation \( q \) is \( \langle a, \{b, c\} \rangle \).

The following example of an \( \mathcal{L} \mathcal{D} \mathcal{L} \) program is included to demonstrate the power of the language (the example is taken from [3], where it is posed as a task for database languages):

**Example 1.1.**

\[
\begin{align*}
\text{part}(P\#, \langle \text{Subpart}\# \rangle) & \leftarrow p(P\#, \text{Subpart}\#). \\
p\text{total\_cost}(X, C) & \leftarrow q(X, C). \\
p\text{total\_cost}(S, C) & \leftarrow \text{partition}(S, S1, S2), \\
& \text{total\_cost}(S1, C1), \\
& \text{total\_cost}(S2, C2), \\
& \text{+(C1, C2, C)}. \\
p\text{total\_cost}(X, C) & \leftarrow \text{part}(X, S), \text{total\_cost}(S, C). \\
p\text{result}(P\#, C) & \leftarrow \text{total\_cost}(\{P\#\}, C). \\
p\text{partition}(S1, S2, S3) & \leftarrow \text{union}(S1, S2, S), \text{intersect}(S1, S2, \{\}). \\
p\text{intersect}(S1, S2, \langle X \rangle) & \leftarrow \text{member}(X, S1), \text{member}(X, S2). \\
p\text{member}(X, S) & \leftarrow \text{union}(\{X\}, S, S). 
\end{align*}
\]

The (normalized) base relations for this problem are \( p(P\#, \text{Subpart}\#) \), which contains tuples of the type (part number, immediate-subpart number), and

---

1 We note that different notations for this operation are possible. For example, the rule might have been written in the following form:

\[
\text{part\_sets}(S\#, P\#$\_set) \leftarrow P\#$\_set = \{P\#: \text{supplier}(S\#, P\#)\}.
\]

or even, as suggested by one of the referees, in the form

\[
p\text{part\_sets}(S\#, \{P\#: \text{supplier}(S\#, P\#)\}).
\]

The \( \langle \cdots \rangle \) notation is used in \( \mathcal{L} \mathcal{D} \mathcal{L} \) because it can be generalized to multiple and nested groupings in one rule. Although we essentially do not deal with such grouping in this paper, we use throughout the \( \mathcal{L} \mathcal{D} \mathcal{L} \) notation.

2 For simplicity we assume that a part cannot have more than one identical immediate subpart.
$q(P\# , Cost)$, which contains tuples of the type (elementary-part number, cost of part). The result is the derived relation $\text{result}(P\#, Cost)$, which contains for each part, elementary or aggregate, the cost of that part. The cost of an aggregate part is the sum of the costs of its immediate subparts.

To see how this result is derived, let us consider the components of the program. The first line derives the $\text{part}$ relation which groups for each part number $P\#$ the set of immediate subpart numbers $\langle \text{Subpart}\# \rangle$. Thus, if the base relation $p$ consists of $p(1, 2), p(1, 7), p(2, 3), p(2, 4), p(3, 5), p(3, 6)$, then the corresponding $\text{part}$ relation will consist of $\text{part}(1, \{2, 7\}), \text{part}(2, \{3, 4\}), \text{part}(3, \{5, 6\})$.

The derived relation $\text{total-cost}$ contains the total cost $C$ for a set of constituent part numbers $S$. The first $\text{total-cost}$ rule derives for each elementary part $X$ a singleton set containing that part, and its cost. The second $\text{total-cost}$ rule is the recursive derivation of the cost of a set of parts as the sum of the costs of its members. The basis of the recursion is the set of those tuples of the $\text{total-cost}$ relation which consist of singleton sets, i.e., elementary-part numbers or aggregate-part numbers. The $\text{partition}$, $\text{intersect}$, and $\text{member}$ rules are auxiliary derivations, in which it is assumed that the only primitive is the $\text{union}$ predicate. The third $\text{total-cost}$ rule derives for each aggregate part $X$ a singleton set $\{X\}$ and the cost $C$ of its immediate subparts. It uses the cost of the set of the immediate subparts, as computed by the previous rule. If base relation $q$ contained $q(4, 20), q(5, 10), q(6, 15), q(7, 200)$, then this $\text{total-cost}$ would contribute the following tuples:

$$\{\text{total-cost}(\{3\), 25\})}, \text{total-cost}(\{2\), 45\})}, \text{total-cost}(\{1\), 245\})\}.$$

As illustrated in the example, the various set predicates and operators are not independent; rather, some can be expressed in terms of others. While a language designer may opt for including many operators in his/her language as primitives, for our purposes it is better to concentrate on a small collection of operators. For most of the formal discussion in this paper, we use only membership and grouping, although we occasionally consider other operators. The reason for this choice is that grouping is a powerful operator: First, other constructors can be expressed using membership and grouping. Second, the problems addressed in this paper are caused by the use of grouping, and they do not occur when other, weaker operators are used.

The power of grouping lies in the fact that it is not a local operation. If a rule contains a grouping operator, we have to consider all substitutions for one of the arguments, for a given fixed substitution for the other arguments, to generate one tuple for the result. Thus, the operation implicitly contains a universal quantifier on a variable (or argument) in the body of a rule. To apply a grouping rule, the truth value of the body under all substitutions must already be known. Thus, if the head predicate also appears in the body, or more generally, if a predicate appearing in the body depends through other rules on the head predicate, the meaning of the rule may be ill defined. Examples for such cycles are presented in Section 3. These problems are similar to those encountered when negation is

---

3It is interesting to note that, if the base relation $q$ were "impure" in the sense that it also contained cost tuples for some of the aggregate parts, the derivation would still hold, provided that the cost of an aggregate part, as recorded in $q$, was the same as the one computed for that part.
introduced [2, 14, 42, 26]. Our solution is similar to the one proposed in these papers—we require programs to be layered, so such cycles are disallowed.

However, it turns out that this condition by itself is insufficient to guarantee that programs have well-defined semantics. Essentially, grouping is powerful enough to express some of the well-known paradoxes of set theory. The grouping constructor allows one to construct new sets, and if one is not careful, such a set may be "too large" to be an element of any reasonable domain. We deal with this problem by imposing one of several additional conditions on programs. One of these is typing, in the spirit of classical mathematical solutions.

The outline of the paper is as follows. The syntax and basic terminology are described in Section 2. Section 3 deals with semantics. The use of sets as elements means that we cannot use the standard Herbrand universe. We define a class of universes that are used to define interpretations and models. The operational and model-theoretic semantics for rules and programs that use grouping are discussed, and it is shown that both are problematic. There exist programs for which both types of semantics seem to be undefined, and in particular, these programs have no models. Further, the notion of minimality is not well defined. Assuming some intuitive notion of minimality, there exist programs that have several incomparable minimal models.

In Section 4 we proceed to offer a solution to the multiple-incomparable-models problem. For that, we adopt the concept of layering, used in [2, 14, 42, 26], to provide well-defined semantics for programs with negation. We show that it works for programs with grouping, except for the case when no model exists, which is treated separately in Section 5. We also adopt the approach to layered programs proposed in [28, 29], which is based on the notion of preference between models and the concomitant notion of a "perfect" or most preferable model. The main result is an iterated fixpoint semantics for layered programs, and a proof that it produces the unique perfect model for the program (provided that a model for the program exists).

The syntactic restrictions imposed by layering do not guarantee the existence of a model, however. In Section 5 we address this issue and investigate conditions that guarantee the existence of models for the programs. The conditions in this section complement the layering restriction of Section 4.

Section 6 contains a brief description of some syntactic extensions to the language. These extensions do not increase the power of $LDL$. Rather, they are introduced as a convenience to the programmer. In addition we present in this section a comparison of $LDL$ with the LPS language [22, 23], which embodies many features similar to $LDL$, and additional observations on various language features. The conclusion of this paper is presented in Section 7; it contains a discussion of some problems that warrant further research.

2. SYNTAX

We use the notational conventions of logic programming. Variables are denoted by capital letters, e.g., $X, Y, Z$; also, "_" denotes an anonymous variable. Constants are denoted by lowercase letters, e.g., $a, b, c$. We shall use $X$ to denote a vector of variables, and similar notation for vectors of constants or terms. Function and
predicate symbols are denoted by strings of lowercase letters. It will be clear from
the context in each case whether a symbol denotes a function or a predicate.
Associated with each function or predicate symbol is an integer—its arity. The
symbol = denotes the equality predicate, and mem denotes the membership
predicate. Both are built-in predicates. We assume that {} is a constant symbol.

**Definition 2.1. Terms** are defined inductively as follows:

- Constants and variables are terms;
- if f is an n-ary function symbol and t₁, . . . , tₙ are terms, then f(t₁, . . . , tₙ) is a
term.

**Definition 2.2.** If X is a variable, then ⟨X⟩ is a grouping expression.

**Definition 2.3.** A (positive) atom is a formula of the form p(t₁, . . . , tₙ) where the tᵢ
are terms or grouping expressions, and p is a predicate symbol of arity n. A
negative atom is a formula of the form ¬p where p is an atom. A literal is a
positive or a negative atom. A rule (or clause) is a formula of the form

head ← body.

where head is an atom, and body is a (possible empty) conjunction of literals.
(We usually represent such a conjunction as a comma-separated sequence of
literals.) A rule with an empty body is called a fact and is written as head. A
rule containing ⟨⟩ in the head is called a grouping rule. A rule is well formed if
it obeys the following syntactic restrictions:

1. ⟨⟩ can only appear in a rule as a grouping expression, i.e., ⟨X⟩.
2. The body contains no occurrence of grouping expressions.
3. The head contains at most one occurrence of a grouping expression.
4. All the literals in the body of a grouping rule are positive.⁴

A program is a finite set of well-formed rules. A program is positive if none of
its rules has an occurrence of a negative literal in its body.

The intended logical and procedural semantics for the above syntax is defined
formally in the sequel. We briefly sketch here the intended meaning of the set
constructs. mem denotes the standard membership predicate. {} stands for the
empty set. The angular brackets are used to group together, into a set, elements
which satisfy some qualification specified in the body of the rule.

Note that so far we have added to the vocabulary of classical logic-programming
languages only {}, mem, and ⟨⟩. In a practical language, one may want to use a
more extensive set of operations, including e.g., the Boolean operations of union,
set difference, set intersection, and set definition by enumeration. The last, in
particular, seems indispensable for dealing with sets. Enumeration may be intro-

---

⁴This condition is for convenience only. It is easy to convert any program to a program that satisfies
this condition, by adding derived predicates.
duced through another built-in predicate \textit{scons}, where the intended meaning of \textit{scons}(t, S_1, S_2) is that \( S_2 = S_1 \cup \{ t \} \). Thus, \( \{ t \} \) is obtained by \textit{scons}(t, \{ \}, S); \( \{ t_1, t_2 \} \) is obtained by \textit{scons}(t_2, \{ t_1 \}, S); and so on. (The use of \textit{scons} for sets is like that of \textit{cons} for lists.) Defining \textit{scons} as a built-in (partial) function is more convenient, as it allows one to directly use enumerated sets. However, it is clear that the predicate and the function provide the same expressive power. Every occurrence of an enumerated term in a rule can be replaced by a new variable, and an instance of \textit{scons} relates it to its constituents.

We show (informally, since we have not yet defined semantics formally) that \textit{scons} can be defined in terms of membership and grouping. The following program defines \( X = S \cup \{ T \} \):\footnote{Note that \textit{scons} as defined here might return a result even if \( S \) is bound to an element that is not a set. This is why \textit{mem}(\_, S) \) has been added to the body of the first rule, to ensure that \( S \) is a set. Also note that this definition allows \( S \) to be an infinite set. The case where \( S \) is the empty set is considered explicitly.}

\[
\begin{align*}
\text{p}(T, S, T) & \leftarrow \text{mem}(X, S). \\
\text{scons}(T, S, \langle X \rangle) & \leftarrow \text{p}(T, S, X), \text{mem}(\_, S). \\
\text{scons}(T, \emptyset, \langle X \rangle) & \leftarrow \text{p}(T, \emptyset, X).
\end{align*}
\]

(Note that \( \_ \) denotes an anonymous variable.) In a similar vein, union can be defined in terms of \textit{mem} and \( \langle \_ \rangle \) as follows:

\[
\begin{align*}
\text{p}(T, S, X) & \leftarrow \text{mem}(X, S). \\
\text{p}(T, S, X) & \leftarrow \text{mem}(X, T). \\
\text{union}(T, S, \langle X \rangle) & \leftarrow \text{p}(T, S, X), \text{mem}(\_, T), \text{mem}(\_, S). \\
\text{union}(\emptyset, S, S) & \leftarrow \text{mem}(\_, S). \\
\text{union}(T, \emptyset, T) & \leftarrow \text{mem}(\_, T). \\
\text{union}(\emptyset, \emptyset, \emptyset).
\end{align*}
\]

Since we allow negation, difference is also expressible. There is therefore no loss of generality in dealing only with \( \{ \}, \text{mem}, \) and \( \langle \_ \rangle \). In the sequel, we use set enumeration in examples; in all these cases, the program segment above that defines \textit{scons} is implicitly assumed. However, it is shown in Sections 4 and 5 that once restrictions on programs are introduced, there are benefits to using \textit{scons} and other constructs as built-in features of the language.

3. INTERPRETATIONS, MODELS, AND RULE APPLICATIONS

It is well known that the semantics of logic programs can be defined in several equivalent ways, e.g., procedural, model-theoretic, and through lattice-theoretic fixed points. In this section we define interpretations and models for rules and programs, and we define the meaning of applying a rule to a set of facts. We also show that various well-known properties of Horn-clause programs fail to hold for our language, thus preventing a straightforward extension of such notions to the semantics of our language.
3.1. Interpretations and Models

We start by defining interpretations and models. Traditionally, semantics for logic programs are based on Herbrand interpretations and models. However, since we allow sets to be arguments of predicates, we need to extend the definition of the universe, so that sets of elements are also elements of the universe.

**Definition 3.1 (Universe of programs).** For a set $S$ let $\text{power}(S)$ denote the set of all subsets of $S$. For each (possibly infinite) ordinal $\alpha > 0$, the extended $\alpha$-universe $U_\alpha$ is defined as follows:

1. For $\alpha = 0$, let $U_0$ be the set of all variable-free terms that do not contain the constant $\{\}$. That is, $U_0$ is the classical Herbrand universe.\(^6\) Note that $U_0$ may be infinite, since the language includes function symbols.

2. If $\alpha$ is a successor ordinal, then we define

   $$G_\alpha,0 = U_{\alpha-1} \cup \text{Power}(U_{\alpha-1}),$$

   $$G_\alpha,i = G_\alpha,i-1 \cup \{f(e_1, \ldots, e_n) : f \text{ is a function symbol and } e_i \in G_\alpha,i-1\},$$

   $$U_\alpha = \bigcup_{j=0}^{\infty} G_\alpha,j.$$

3. If $\alpha$ is a limit ordinal (different from 0), we let

   $$U_\alpha = \bigcup_{i<\alpha} U_i.$$

The need to have sets as elements in our universe was explained briefly above. The universes we have constructed, one for each ordinal, are the natural extensions of the notion of a Herbrand universe. These universes will serve as the domains of interpretations of programs. Note that an interpretation for a program cannot have a domain that is closed under the $\text{Power}$ operation, since such a domain is not a set.

An interpretation of a program resembles a Herbrand interpretation. One significant difference is that the membership predicate, in addition to the equality predicate, is interpreted, i.e., it is assigned its fixed, standard set-theoretic meaning; similarly the constant $\{\}$ is assigned to $\emptyset$, the empty set.

**Definition 3.2 (Interpretation).** An interpretation is defined as follows [where $I(t)$ denotes the interpretation of $t$]:

1. $U_\alpha$ is the domain of interpretation, for some $\alpha > 0$.

2. Constants are assigned to themselves in $U$, i.e. $I(c) = c$. However, the constant $\{\}$ is assigned to $\emptyset$, the empty set, which is a member of $U_\alpha$.

\(^6\)More precisely, $U_0$ and the universe that is constructed from it depend on the choice of the set of constants. If we start only with the constants of a given program, then $U_0$ is the Herbrand universe. We choose to start from the set of all constants that can be used in the language. Our results hold also for every universe whose set of constants contains those of a given program.
(3) If \( f \) has arity \( n \), then \( f \) is assigned a mapping from \( U_a^n \) to \( U_a \) that maps \((e_1, \ldots, e_n)\) into \( f(e_1, \ldots, e_n) \), where \( e_i \in U_a \) for \( i = 1, \ldots, n \).

(4) For each predicate symbol of arity \( n \), except for \( = \) and \( \text{mem} \), there is an assignment of a relation on \( U_a^n \).

(5) \( = \) is assigned the standard interpretation of the equality predicate. For any two elements \( a, b \) of \( U_a \), \( a = b \) iff \( a \) and \( b \) are the same element. (As an example, \( \{a, \{a, \{a, b\}\}\} = \{a, \{\{a, b\}, a\}\} \).

(6) \( \text{mem} \) is assigned to the standard set-theoretic interpretation of membership. That is, \( \text{mem}(e, S) \) iff \( S \) is a set, \( e \) is a member of \( S \), and both \( e \) and \( S \) are in \( U_a \).

From now, we usually omit the subscript \( \alpha \); \( U \) stands for an arbitrary \( U_a, \alpha > 0 \).

**Definition 3.3.** A \( U \)-fact is an object of the form \( p(e_1, \ldots, e_n) \) where \( p \) is an \( n \)-ary predicate symbol (but not a built-in predicate) and \( e_i \in U \) for \( i = 1, \ldots, n \). For a set \( M \) of \( U \)-facts, the extension of a predicate \( p \) in \( M \) is the subset of \( M \) of facts that have \( p \) as their predicate symbol; this is denoted as \( \text{ext}(M, p) \).

In traditional logic programming, a subset of the \( U \)-facts defines an interpretation. Similarly, a subset of the set of \( U \)-facts defines an interpretation in the extended universe \( U \), since the interpretations of constants, functions, and built-in predicates are fixed. An interpretation \( I \) is essentially a pair \((U_a, M)\), for some \( \alpha \) and a set \( M \) of \( U \)-facts. In the sequel, the terms “interpretation” and “set of \( U \)-facts” are used interchangeably.

### 3.2. Model Based Semantics

We can now define the notion of truth value for a rule, viewed as a formula, induced by an interpretation \( I \). The definitions are essentially as in [25], with the extensions needed for the treatment of the membership predicate and of the grouping operator.

**Definition 3.4.** A binding \( \theta \) is a set of pairs
\[
\{X_1/e_1, \ldots, X_n/e_n\}
\]
where \( X_1, \ldots, X_n \) are variables, and \( e_1, \ldots, e_n \) are elements of \( U \). If a binding \( \theta \) is defined for all the variables of a term \( t \), then it maps it to an element of \( U \), denoted \( t\theta \), obtained by the simultaneous replacement of all the variables of \( t \) by the corresponding elements in \( U \). If a binding is defined for all variables in a literal \( A \), then it maps the literal to a \( U \)-fact, or the negation of a \( U \)-fact, denoted \( A\theta \), obtained by simultaneous replacement of all variables by their images under \( \theta \). Similarly, a binding that is defined for all variables in a rule without grouping, \( r \), maps it to another rule that contains only \( U \)-facts, denoted \( r\theta \). Such an image of a rule is called a rule instance. For a rule with grouping, the binding maps the body to an instance of the body, and it maps the terms in the head to elements of \( U \). However, the binding \( \theta \) does not apply to the grouping expression; e.g., \( \langle X \rangle \) is left unchanged.
From the definitions above, we obtain immediately a definition of the truth value of a rule without grouping, viewed as a formula, under a binding $\theta$. Recall that an interpretation is defined by a set of $U$-facts. The image under $\theta$ of a rule is true in a set $M$ of $U$-facts if either the image of the head predicate under $\theta$ is in $M$, or the body evaluates to false. A body evaluates to false if either

1. it contains a positive (respectively, negative) literal involving $=$ or $\text{mem}$ such that the predicate does not hold (respectively, holds) on its arguments in $U$ [note that $\neg \text{mem}(e, S)$ is true if $S$ is not a set], or

2. for some literal, with predicate symbol other than $=$ and $\text{mem}$, the literal is positive and its image under $\theta$ is not in $M$, or the literal is negative and the complement of its image is in $M$.

Since rules are assumed to be universally closed, a rule is true in a set $M$ if it is true under all bindings that are defined on all its variables. Note that negation can be applied to any atom. The interpretation of $\neg p$ is the complement of $p$, so $p$ is true iff $\neg p$ is false. The image under $\theta$ of a rule $r$ with respect to $M$ is false if it is not true.

The grouping operator requires a special definition, since it implicitly involves universal quantification. We first explain its meaning informally.

Consider first the formula

$$p(\langle X \rangle) \leftarrow \text{body}(\bar{Z}, X).$$

where $\text{body}(\bar{Z}, X)$ is a conjunction of literals $B_1, \ldots, B_m$. Suppose that there is a nonempty set of bindings for which $\text{body}(\bar{Z}, X)$ is true under $I$. Then the formula is true if $p$ holds on the element of $U$ which is the set of all values assigned by these bindings to $X$, and the formula is false otherwise. However, when the set of elements to be grouped is empty, the formula is true even if $p$ does not hold on the empty set. This (seemingly arbitrary) decision is motivated by the common use of grouping in database systems, where only nonempty sets are grouped. (It is similar to the interpretation of implication in logic, where an implication is true when the body is false. We should stress that in the presence of grouping, the symbol $\leftarrow$ can no longer be thought of as simple implication, because it is a "global" construct that looks at many substitutions. See also below for yet another reason.)

Generally, consider a formula of the form

$$p(\bar{T}, \langle X \rangle) \leftarrow \text{body}(\bar{Z}, X)$$

where $\bar{T}$ is an $n$-tuple of terms, involving the set of variables $\bar{Y}$, and $\bar{Z}$ are the variables appearing in the body except for $X$. ($\bar{Y}$ may however include $X$; further, not every member of $\bar{Y}$ needs to appear in the body.) Intuitively, one can view the body as evaluating $(n + 1)$-ary relations $R$ and $R'$, defined as follows. For each binding for which the body is true, a tuple consisting of the values of the terms in $\bar{T}$ in the first $n$ columns, and of the value of $X$ for that binding in the last column, is put in $R$. Then, $R$ is partitioned "horizontally", for each distinct combination of values in $\bar{T}$. The values for the $X$-column in each element of the partition are grouped into a set, thus creating $R'$, which has the same number of columns, but with sets in the last column. The rule is true for a given binding, for which the body is true, if $p$ holds on the associated tuple of $R'$. (Recall that each row of $R'$
is associated with a set of rows of $R$, and each row of $R$ is associated with a binding.) The rule is therefore true if $p$ holds on each tuple of $R'$, i.e., if the extension of $p$ in $M$ contains $R'$. Otherwise, the rule is false.

As discussed above, if for some binding there is no compatible tuple in $R$, i.e., the grouped set is empty, then the rule is true that for binding. In addition to the reasons listed above, note that the alternative, namely requiring the extension of $p$ to contain a tuple $(\vec{T}, \varnothing)$ whenever the body of the rule is false under $\theta$, would require the extension of $p$ to contain a “very large” set of tuples, making programs that try to group on $p$ undefined. (This problem can be avoided if the body of the rule defining $p$ is added to any rule with grouping whose body contains an occurrence of $p$, but that would make programs unnecessarily complex.) Note that in the case that $X \in Y$, the grouped set is a singleton; this is allowed, although we do not expect it to occur often in practice.

**Definition 3.5.** The truth value of a rule with grouping is defined formally as follows. Let $r$ be the rule

$$p(\vec{T}, \langle X \rangle) \leftarrow \text{body}(\vec{Z}, X)$$

where $p$, body, $\vec{T}$, $\vec{Z}$, $X$ are as above, and let $M$ be a set of $U$-facts. A binding $\theta$ is applicable to $r$ w.r.t. $M$ if it is defined for all variables of $r$ and, in addition, it satisfies where $B_i$ are the literals in body($\vec{Z}$, $X$):

- If $B_i$'s predicate symbol is $\text{mem}$ or $=$ and $B_i$ is a positive (respectively, negative) literal, then $B_i$ holds (respectively, does not hold) on its arguments in the standard set-theoretic interpretation of $=$ and $\text{mem}$ (see Definition 3.2).
- Otherwise, if $B_i$ is a positive (respectively, negative) literal whose predicate symbol is different from $\text{mem}$ and $=$, then $B_i \in M$ (respectively, $B_i \notin M$) for all $i$.

Denote by $\Sigma$ the set of bindings applicable to the rule w.r.t. $M$. Define an equivalence relation $\equiv$ on $\Sigma$, viz., $\theta_1 \equiv \theta_2$ if for all $t \in \vec{T}$, $t \theta_1 = t \theta_2$ (equality in $U$). Let $S\Sigma$ be the set of equivalence classes under $\equiv$. Let $\Sigma_j$ be an equivalence class under $\equiv$, i.e., $\Sigma_j \in S\Sigma$, and let $\theta_j$ be an arbitrary representative of $\Sigma_j$. We say that the rule is true under a binding $\theta$ in $(U, M)$ if $\theta$ is not in $\Sigma$, or else, if $\theta \in \Sigma_j$, then $p(\vec{T}, \{X: \theta \in \Sigma_j\})$ is in $(U, M)$. The rule is true in $(U, M)$ if it is true under all bindings.

We note that for a given $U_a$ as the domain of interpretation, and for a class $\Sigma_j$, $(X: \theta \in \Sigma_j)$ is well defined; however, it need not be an element of $U_a$. This problem persists, no matter how large $\alpha$ is. If the set is not in $U_a$, then the truth value of the rule is undefined for that interpretation. Otherwise, if the truth value of the rule is neither true nor undefined, then the rule is false. In this paper, the notion of a false rule is not used in ascribing semantics to programs. This issue is addressed again later.

7Alternatively, we could say that the rule evaluates to true precisely when the following (second-order) formula is true in $M$:

$$\forall \vec{Z} \forall X \forall [\text{body}(\vec{Z}, X) \rightarrow \exists S[p(\vec{T}, S) \land \forall X'(\text{body}(\vec{Z}, X') \leftrightarrow X' \in S)]].$$
Definition 3.6. An interpretation $I$ is a model for a set of rules $R$, i.e., for a program or a program segment, if each rule in $R$ is true in $I$.

Example 3.1. As mentioned above, the set-enumeration expression $h({1})$ used below, although legal in L, is not legal in the language we have defined. However, as shown, it can be replaced by a set of rules that use grouping that define this fact. Let the program be

$$q(X) \leftarrow p(X), h(X).$$
$$p(\langle X \rangle) \leftarrow r(X).$$
$$r(1).$$
$$h(\langle 1 \rangle).$$

The set $\{r(1), h(\langle 1 \rangle), p(\langle 1 \rangle), q(\{1\})\}$ is a model, whereas $\{r(1), h(\langle 1 \rangle), p(\langle 1, 2 \rangle)\}$ is not a model.

We note that we are considering only interpretations and models of a certain type. First, we require that constant symbols are to be assigned to distinct constants, and functions remain uninterpreted, i.e., $f(t_1, \ldots, t_n) = g(s_1, \ldots, s_n)$ iff $f$ and $g$ are the same function symbol, and $t_i = s_i$ for $i = 1, \ldots, n$. This restriction is essentially the same as that used to define Herbrand interpretations and models in the classical theory of logic programming. Second, we restrict the interpretation of $\text{mem}$ to be the standard (mathematician's) membership, i.e., we interpret sets as in mathematics. It is possible, of course, to consider arbitrary interpretations, in which case we will be dealing with arbitrary models of set theory. One is then tempted to ask whether a parallel of Herbrand's theorem might be proved. We indicate in Section 7 why we do not believe this is possible. We also discuss there why a proof theory does not exist for our language. In this paper, we restrict our attention to models in some $U_n$, as defined above, and "model" will be used with this meaning only.

In the definition of grouping, Definition 3.5, the empty set is not grouped. However, this does not lead to any loss of expressive power. We still have the ability to simulate grouping of the empty set, as illustrated by the following example.

Example 3.2. Given relations $\text{human}(X)$ and $\text{student}(\text{Name}, \text{Degree})$, we are required to define a relation $\text{stud_set}(X,Y)$ such that $Y$ is the set of degrees of the human $X$. If the human in question has no degrees, we want to associate the empty set with $Y$ in the corresponding tuple in $\text{stud_set}(X,Y)$. The following program defines the relation $\text{stud_set}$:

$$p(X, \langle Y \rangle) \leftarrow \text{human}(X), \text{student}(X,Y).$$
$$\text{stud_set}(X, \{\}) \leftarrow \text{human}(X), \neg p(X,Y).$$
$$\text{stud_set}(X,Y) \leftarrow p(X,Y).$$

3.3. Procedural Semantics

To define procedural semantics for programs, the first step is to define the meaning of applying a rule to a set of facts.
Definition 3.7. The procedural meaning of a rule without grouping is defined as follows. Let \( r \) be a rule
\[
B \leftarrow B_1 \cdots B_m.
\]
Let \( M \) be a set of \( U \)-facts. The application of \( r \) to \( M \), denoted \( r(M) \), is defined as
\[
r(M) = \{ B\theta : \theta \text{ is applicable to } r \text{ w.r.t. } M \}.
\]
The procedural meaning of a rule with grouping is defined as follows. Let \( r \) be the rule
\[
P(T,(X)) + \text{body}(Z,X).
\]
where \( p \) (the head of the rule), body, \( T \), \( Y \), \( Z \), and \( X \) are as in the previous section; let \( M \) be a set of \( U \)-facts, and let \( SE = \{ Z_j \} \) and \( \theta_j \) be as defined there. Define
\[
p\Sigma_j = p(\bar{T}\theta_j, \{ X\theta : \theta \in \Sigma_j \}).
\]
Then
\[
r(M) = \{ p\Sigma_j : \Sigma_j \in SE \}.
\]

As noted previously, for a class \( \Sigma_j \), \( \{ X\theta : \theta \in \Sigma_j \} \) is well defined; however, it need not be an element of \( U \). If it is not, \( p(\bar{T}\Sigma_j, \{ X\theta : \theta \in \Sigma_j \}) \) cannot be a \( U \)-fact, and \( r(M) \) is undefined. This issue is discussed later.

Given a program \( P \) and a set \( M \) of \( U \)-facts, intuitively, the application of \( P \) to \( M \) is the set of facts obtained by applying the rules of \( P \) to \( M \), adding the results to \( M \), and repeating until no new facts can be generated. For regular Horn-clause programs, this process is known to produce a unique result, regardless of the order of application of the rules. It is known that this is not the case when the bodies of the rules contain negated predicates. We will show shortly that this problem arises also in the presence of grouping, even without negation. A solution will be proposed later.

We intend our language (or practical languages based on it) to be used to run finite programs on finite databases, and obtain finite results. Nevertheless, in considering its semantics, we would like to assign a meaning also to those programs that generate infinite extensions for some predicates. Next, we examine whether it is possible to restrict attention to \( U_\alpha \) for some fixed, "small" \( \alpha \).

One would like to claim that for some fixed \( \alpha \), all the programs have a well-defined meaning in \( U_\alpha \). Note that this is the case for programs without grouping and negation, since they have a meaning in \( U_0 \). Now that we have a reasonable understanding of the meaning of programs, we can show that choosing, e.g., \( \alpha \) to be \( \omega \) is too restrictive. The following example demonstrates the need to iterate in the construction of \( U \) to an ordinal beyond \( \omega \), even when a finite

---

\(^8\)The problem of a possibly infinite result exists in languages that allow the use of function symbols, even without sets. Since whether the result of a program is infinite is in general undecidable, it is not appropriate to assign meaning only to programs that generate a finite answer, as that would mean that whether a program has a well-defined semantics is undecidable.
database is assumed. We observe that grouping allows us to collect the extension of a predicate into a set, and assert a predicate over this set. This can be repeated many times, so we need a universe that contains sets, tuples with sets as elements, sets of such tuples, and so on. \( U_\omega = \bigcup_{i=0}^{\infty} U_i \) is such a universe. However, \( U_\omega \) is inadequate.

**Example 3.3.** Consider the following program:

\[
\begin{align*}
q(\langle X \rangle) & \leftarrow q(X). \\
p(\langle X \rangle) & \leftarrow q(X). \\
q(a). \\
\end{align*}
\]

The first rule generates \( q \)-facts with finite but unbounded nesting, with a seed provided by the third rule. The second rule collects all these terms into one set, and asserts \( p \) over this set. Clearly, the nesting depth of this set is not bounded by any integer. However, this set is not a member of \( U_n \), since each member of \( U_\omega \) is also a member of \( U_n \), for some finite \( n \), so its nesting depth is bounded by \( n \).

Conditions on programs that rule out programs like the one in Example 3.3 and allow us to restrict attention to \( U_\omega \) are considered in Section 5.

### 3.4. Problems with the Classical Approach to Semantics of Programs

For classical logic programs, the following well-known properties hold: each program has a (Herbrand) model; the intersection of models is a model, and hence the program has a unique minimal Herbrand model; this unique minimal model is also the least fixed point of the program, viewed as an operator on sets of facts. If negation is allowed in bodies of rules, then minimal models still exist, but uniqueness is not guaranteed. Similarly, the procedural meaning of a program is not well defined, since different sequences of rule applications to a set of facts yield different results. A solution, proposed in \([2,14,42,26]\), is to restrict attention to the so-called stratified (or layered) programs. For such programs, it is shown that there exists a minimal model that can be considered as the intended meaning of the program.

Since our language allows negation, the same problems exist. However, similar and even more severe problems arise because we use sets and grouping. We show that, even without using negation, a program may have several minimal models, or worse, it may have no model. We also show that models are not closed under intersection, even when intuitively we feel that they are comparable, and further, that it is not clear how to compare models. Similar problems exist for the procedural semantics of programs.

**Example 3.4.** Consider the program \( P \):

\[
\begin{align*}
p(\langle X \rangle) & \leftarrow q(X). \\
q(2). \\
\end{align*}
\]
Possible models for the program are

\[
A = \{q(1), q(2), p(\{1, 2\})\}, \\
B = \{q(2), q(3), p(\{2, 3\})\}, \\
C = \{q(1), q(2), q(3), p(\{1, 2, 3\})\}, \\
D = \{q(2), p(\{2\})\}.
\]

However, \(A \cap B\) is not a model, as it does not contain \(p((2))\). Similarly, \(A \cap C\) and \(B \cap C\) are not models. Note that, intuitively, \(C\) is "larger" than \(A\) and \(B\), which are both larger than \(D\), but \(D\) is not a subset of either \(A\), \(B\), or \(C\).

The idea of performing setwise intersections of set elements to produce new "smaller" models also fails. The intersection of \(\{1, 2\}\) and \(\{2, 3\}\) in the previous example does indeed produce \(\{2\}\), so in this example we might say that \(D\) is the "generalized intersection" of \(B\) and \(C\). However, consider what happens if we add facts \(r(1), r(3)\) and a rule \(p(X) \leftarrow r(X)\) to the program!

Next, a positive \(\mathcal{KB}\) program does not necessarily have a unique minimal model.

**Example 3.5.** Consider the program \(P:\)

\[
p((X)) \leftarrow q(X).
q(Y) \leftarrow w(S, Y), p(S).
q(1).
w(\{1\}, 7).
\]

Note, first, that \(M = \{q(1), w(\{1\}, 7)\}\) is not a model; since the body of the first rule holds for \(X = 1\), \(p\) must hold on some set. Note that even if we add \(p(\{1\})\) to \(M\), it is still not a model, since then also \(q(7)\) should hold. However, \(M_1 = M \cup \{q(2), p(\{1, 2\})\}\) is a model. \(M_2 = M \cup \{q(3), p(\{1, 3\})\}\) is also a model. It can be checked that \(M_1\) and \(M_2\) are minimal, by the classical definition of minimality based on set inclusion. Intuition supports this claim of minimality, since no facts can be removed from either one, nor can any elements be removed from set-valued facts, without making the result a nonmodel. So \(P\) does not have a unique minimal model.

As for procedural meaning of the program, note that if we start from the empty set of facts, we first have to include at least one of the two facts of \(P\). Once \(q(1)\) is in, the first rule can be applied to generate \(p(\{1\})\). Once \(w(\{1\}, 7)\) has also been included, we can apply the second rule to obtain \(q(7)\). Then the first rule can again be applied to generate \(p(\{1, 7\})\). No more rules can be applied now. The result is a model, but it is not minimal—\(p(\{1\})\) can obviously be removed. Note that the fact \(p(\{1\})\) was needed for the application of the second rule to generate \(q(7)\). Indeed, the minimal model \(M_3 = \{q(1), q(7), w(\{1\}, 7), p(\{1, 7\})\}\) cannot be generated by applying the rules in any order to the empty set of facts.

Another example where applications of rules in different order generates different sets of facts is Example 3.3. There, it is possible to apply the rule...
$p(\langle X \rangle) \leftarrow q(X)$ after any finite number of applications of the rule $q(\langle X \rangle) \leftarrow q(X)$. Different sets of facts may be generated, depending on when the first rule is applied. Note that the intuitive intended semantics is, probably, that the rule for $p$ should be applied only after the rule for $q$ cannot be applied any more, that is, after the least fixed point for the second rule has been reached. Thus, the fact $p((a, (a), \ldots))$ can only be generated by a transfinite sequence of rule applications. This is another contrast to the classical theory.

Our next two examples demonstrate that not all programs have models.

**Example 3.6.** Consider the program $P$:

$$p(\langle X \rangle) \leftarrow p(X).$$
$$p(1).$$

Suppose $P$ has a model $M$. Let $Z = \{e : e \in U \land p(e) \in M\}$. Since $M$ is a model of $P$, the first rule must be true in $M$, so $p(Z)$ must be a fact in $M$. From that and the definition of $Z$ it follows that $Z \in Z$—a contradiction. Thus $P$ does not have a model. This is reminiscent of the Russell-Whitehead paradoxes. Note that the problem does not depend on the specific choice of $U$. A procedural interpretation of $P$ is also problematic. Starting from the empty set, we will be forced to add an infinite sequence of facts, $p(1), p((1)), p((1, (1))), \ldots$. This is the only order in which the facts can be generated, and no matter how far we continue, even using transfinite induction, we will not obtain a model of $P$.

**Example 3.7.** Consider the program $P$:

$$q(\langle X \rangle).$$
$$p(\langle X \rangle) \leftarrow q(X).$$

In the universe $U$, or in any other universe, $q$ holds for all elements of the universe. Hence, if the set for which $p$ must hold to satisfy the second rule is an element of the universe, then the universe must contain itself as an element.

### 3.5. Liberal and Conservative Semantics

As illustrated by the examples, a simple extension to our language of the standard approach to the semantics of logic programs is fraught with problems. Two major problems, and a few minor ones, have been identified. First, the grouping operator involves an implicit universal quantifier. When a predicate is defined in terms of itself, either directly (as in Examples 3.5 and 3.6) or indirectly, and the definition involves grouping, the use of the grouping operator may be ill defined. A similar problem exists in applying negation in the bodies of rules. In contrast, cyclic definitions in positive programs without grouping pose no problem.

Intuitively, the problem can also be viewed as follows. Negation, or grouping, may be applied at different points of computations, so different sequences of rule applications lead to different models. (Recall, however, that some minimal models cannot be constructed by any sequence of rule applications. The reason is that
grouping is a nonmonotonic operator. Adding some facts to the extension of a predicate makes a previously computed result of grouping application redundant. This is reflected in a model, but is not reflected in a set of facts constructed by a sequence of rule applications.

Our solution to this problem, presented in the next section, is similar to that advocated for negation. We require programs to be layered. Layering imposes an implicit order on rule applications, and thus determines a unique model.

The second problem is that a model for a program does not always exist, and for such a program it is usually the case that procedural semantics is also not defined, since for some application of a grouping rule, the resulting set has “too many” elements. As illustrated in Example 3.7, this problem is not solved by layering. (An intuitive understanding of layering suffices for this observation.) The problem is related to the expressive power of the language. A separate, independent solution is needed. Our approach follows classical mathematical solutions to such problems, and is presented in Section 5.

We note now some minor problems. First, we have seen that even for layered programs, the fixpoint is not guaranteed to be reached after at most \( \omega \) steps. Therefore, we will use transfinite constructions. Another problem is that it is not clear how to compare models. This issue is treated only briefly in this paper.

Before we proceed to the presentation of layering, we need to address a technical issue. The result we intend to prove in the next section has the form: if the program is layered, the model-based and the procedural semantics are well defined and equal. Unfortunately, in this form it is not true, because of the second problem discussed above, namely, that a model for a program does not always exist. Our solution is to introduce below a “more liberal” semantics, in which models are always defined, and grouping can always be applied. We prove the results of the next section for this semantics. We emphasize that this is only a technical device that allows us to state and prove results in a simple way. We are not proposing this semantics to be used in practice. The results of the next section hold, in particular, also for programs that satisfy the restrictions of Section 5, and for such programs the liberal semantics is identical to the standard semantics, as defined previously in this section.

Recall the definitions of truth value and of the procedural meaning of a rule with grouping. Let the rule \( r \) be

\[
p(\overline{T}, \langle X \rangle) \leftarrow \text{body}( \overline{Z}, X).
\]

where \( \overline{T}, \overline{Y}, \overline{Z}, \) and \( X \) are as before, and let \( M \) be a set of \( U \) facts. As explained informally above, given the rule, we construct a relation \( R' \) whose typical tuple contains the values of \( \overline{T} \) for a binding, and the set of \( X \)-values that correspond to this vector \( \overline{T} \) of values. Then \( r \) is true if the extension of \( p \) in \( M \) contains \( R' \). The problem is that some of the sets appearing in the last column of \( R' \) may be “too large” in the sense that they are not elements of \( U \). Recall the definitions of \( \Sigma, \Sigma_j, \theta_j, \) and \( p \Sigma_j \). Let us now redefine

\[
p\Sigma_j = \begin{cases} p(\overline{T}\theta_j, \{ X\theta : \theta \in \Sigma_j \}) & \text{if } \{ X\theta : \theta \in \Sigma_j \} \in U, \\ \text{undefined} & \text{otherwise}. \end{cases}
\]
Given this revised definition, we offer the following two alternative procedural meanings for the application of a rule \( r \) to a set \( M \), and correspondingly, two alternative model-based semantics:

**Definition 3.8.**

I. Procedural semantics:

- **\( P_1 \) (Conservative):** \( r(M) = \{ p\Sigma_j : \Sigma_j \in SE \} \), provided that \( p\Sigma_j \) is defined for all \( j \), and \( r(M) \) is undefined otherwise.
- **\( P_2 \) (Liberal):** \( r(M) = \{ p\Sigma_j : \Sigma_j \in SE, p\Sigma_j \) is defined\}.

II. Model-based semantics:

- **\( M_1 \) (Conservative):** \( r \) is true in \( M \) if for all \( j \), \( p\Sigma_j \) is defined and is in \( M \). If some \( p\Sigma_j \) is undefined, the truth value of \( r \) in \( M \) is undefined.
- **\( M_2 \) (Liberal):** \( r \) is true in \( M \) if for all \( j \) such that \( p\Sigma_j \) is defined, \( p\Sigma_j \) is in \( M \).

We note that it is reasonable only to associate \( P_1 \) with \( M_1 \), and \( P_2 \) with \( M_2 \). Indeed, for any of the two mixed combinations, the application of a rule repetitively to a set \( M \) will not necessarily produce a model in which the rule is true. We refer to the combination of \( P_1 \) and \( M_1 \) as the *conservative* semantics, and to the combination of \( P_2 \) and \( M_2 \) as the *liberal* semantics. Although the liberal semantics allows us to apply rules and assign truth values in all cases, it is not really a solution to the problem of "too large" sets. For the program of Example 3.7 where \( q \) holds for all elements and \( p \) groups on \( q \), it would assign the rule the value *true*, although in the interpretation relation \( p \) is empty and \( q \) holds for all elements. What we really want is to restrict attention to programs in which such anomalies do not occur and for each \( \Sigma_j \), \( p\Sigma_j \) is defined whenever a rule is applied. That is, we want to deal only with programs that have well-defined (procedural and model-based) conservative semantics. Conservative semantics is the natural extension of the standard classical semantics of logic programs. Restrictions on programs guaranteeing that the conservative semantics is well defined are presented in Section 5. The liberal semantics, as already explained, is a temporary device that enables us to obtain the results of Section 4 without the need to qualify them by preconditions about the validity of grouping applications.

### 4. SEMANTICS OF PROGRAMS

Our goal is to show that under suitable conditions, programs have well-defined model-theoretic and procedural semantics. In this section, we describe syntactical restrictions on programs, thereby defining the class of layered programs. The restrictions amount to splitting programs into distinct partitions, called *layers* or *strata*. We show that, under the liberal semantics defined at the end of the previous section, layered programs have well-defined and equivalent model-based and procedural semantics.
4.1. Layered Programs

Consider a program $P$; define relations $\geq$, $\geq$ on the uninterpreted predicate symbols appearing in $P$. (We discuss built-in predicates below.)

- $p \geq q$ if there is a rule in $P$ in which $p$ is the head predicate symbol, there is no occurrence of the form $\langle X \rangle$ in the head, and $q$ appears nonnegated in the body of the rule.

- $p > q$ if there is a rule in $P$ in which $p$ is the head predicate symbol, there is an occurrence of the form $\langle X \rangle$ in the head, and $q$ appears in the body of the rule.

- $p > q$ if there is a rule in $P$ in which $p$ is the head predicate symbol and $q$ appears negated in the body of the rule.

Intuitively, $\geq \cup >$ represents the dependency relationships among the predicates of $P$: $p \left( \geq \cup > \right)^+ q$, where $(\geq \cup >)^+$ denotes the transitive closure of $\geq \cup >$, means that $p$ depends, possibly through several rules, on $q$.

**Definition 4.1.** A program $P$ is *layered* if there is no sequence of predicate symbols in the rules of $P$ of the form

$$p_1 \theta_1 p_2 \cdots \theta_{k-1} p_k,$$

where $\theta_i \in \{\geq, >\}$, for $i = 1, \ldots, k - 1$, such that $p_1 = p_k$, and for some $j$, $1 \leq j \leq k - 1$, $\theta_j$ is $>$. An alternative definition may be obtained as follows. Define $p \gg q$ if there is a sequence of $\geq \cup >$ relationships from $p$ to $q$, such that at least one of them is $>$. Then we have:

**Lemma 4.1.** A program $P$ is layered if and only if the relation $\gg$ is both transitive and irreflexive.

Yet another characterization of layered programs has been used in the literature.

**Definition 4.2.** The *base* predicates of a program $P$ are those that appear in the body of some rule but do not appear in the head of any rule of $P$. All other predicates of $P$ are called *derived*.

Note that base and derived predicates are relative to $P$. By the definition, a predicate that appears in a fact of $P$ is derived, since a fact is a head of a rule without a body. We allow $P$ to contain facts which are occurrences of any of the derived predicates.
Definition 4.3. A layering for $P$ is a partition $L_0, \ldots, L_m$ of the predicate symbols of $P$ such that\footnote{Our definition of layering differs from that of [2, 14, 42, 26] in that they do not require condition (1). The definition we use is more convenient for the treatment of bottom-up evaluation of programs, and it considerably simplifies the constructions of the next section.}

1. $L_0$ contains precisely the base predicates of $P$, and
2. for all $p, q$:
   a. if $p \geq q$, $p \in L_i$, $q \in L_j$, then $i \geq j$, and
   b. if $p > q$, $p \in L_i$, $q \in L_j$, then $i > j$.

A program is layered if there exists a layering for it.

Observe that there may be more than one layering for a given program. (The corresponding concepts used in [2] are stratification and stratified, respectively.)

Lemma 4.2. A program $P$ is layered according to Definition 4.1 if and only if it is layered according to Definition 4.3.

Proof. For the proof, let us temporarily refer to programs that are layered according to Definition 4.1 as admissible, and to those that are layered according to Definition 4.3 as layered.

→: Define $G(P)$ to be a graph whose nodes are the predicates of $P$, and whose edges are the $\geq \cup >$ relationships. Then, if $P$ is admissible, there is no cycle that contains $>$ in $G(P)$. Furthermore, the graph whose nodes are the strongly connected components of $G(P)$ is acyclic. Any topological ordering [21] of that graph is a valid layering.

←: Let $L_1, \ldots, L_m$ be a layering for $P$. Suppose, for the sake of deriving a contradiction, that $P$ is not admissible. It follows from the definition of admissibility that there is a sequence of predicate symbols of $P$

$$p_1 \theta_1, p_2 \theta_2 \cdots p_{k-1} \theta_{k-1} p_k, \quad \theta_j \in \{>, \geq\} \quad \forall j = 1, \ldots, k - 1,$$

such that some $\theta_j$, $1 \leq j \leq k - 1$, is $>$ and $p_1 = p_k$. By the definition of layering, for $i = 1, \ldots, k - 1$, the layer of $p_i$ is higher than or equal to the layer of $p_{i+1}$. Also, the layer of $p_{j+1}$ is lower than the layer of $p_j$. But then $p_1$ must be in a layer strictly higher than the layer of $p_k$, which is a contradiction, since $p_1 = p_k$. □

The following special case will be used in the ensuing proofs. Define a program to be two-layered if it has a layering consisting of two layers. Clearly, a program is two-layered if and only if for all distinct predicates $p$ and $q$, if $p \gg q$ then $q$ is a base predicate.

Let us consider now the role of the built-in predicates, $=$ and $\text{mem}$, in the definitions above. One problem in defining the semantics of programs that contain negation or grouping, or both, is the existence of incomparable models, since it is not clear which model represents the intended semantics. Intuitively, adding a fact to the extension of a predicate in a model, may require that some facts be removed
from the extension of other predicates that depend on it, negatively or by the use of grouping. The resulting model is then incomparable to the original model. Layering assigns priorities to predicates, so that those that appear lower in the layering are to be minimized with a higher priority. A model that satisfies these priorities can be constructed bottom-up, by induction on the layers.

Now, the extensions of built-in predicates, as well as those of their complements, are fixed and cannot be changed. All models agree on those predicates. Hence, for the definitions above, built-in predicates may be considered to be on the lowest layer of every program, i.e., they are base predicates. This also holds for every predicate that is definable in terms of built-in predicates and functions only.

4.2. Comparing Models

In the following discussion, assume that \( U \) is fixed. For a program \( P \) and a set of \( U \)-facts \( M \), we say that \( N \) is a minimal model of \( P \) that contains \( M \) if \( M \subseteq N, N \) is a model of \( P \), and there is no subset of \( N \) that contains \( M \) and is a model of \( P \). Note that the intersection of a decreasing chain of models of \( P \) is a model of \( P \). It follows that if a model of \( P \) exists in \( U \), then it contains a minimal model. Indeed, if it does not contain a minimal model, then it is possible to define a transfinite decreasing chain of models of \( P \). We can associate with each model in the chain an element that is in the model but not in its successor, thus obtaining a transfinite sequence of distinct elements of \( U \), thereby reaching arbitrary ordinals—a contradiction. However, incomparable minimal models may exist, as shown by our previous examples.

Intuitively, layering solves the problem of assigning meaning to programs that contain negation, and it (partially) solves that problem when grouping is also used, since it assigns priorities to the predicates. Predicates with high priorities are those that appear low in the layering, that is, \( q \) has higher priority than \( p \) iff \( p \gg q \). Extensions of the predicates with a higher priority are computed first, and then grouping can be applied to them in the computation of predicates that have lower priorities. The predicates that are computed first have minimal extensions, according to the accepted semantics of simple programs. Those that are computed later have minimal extensions, assuming that the extensions for those predicates that have been previously computed are fixed. Some models reflect these priorities more than others, and intuitively, the intended model of a program is the one that reflects the priorities more than all other models. These ideas have been formalized in [28,29] for programs that contain negation but do not deal with sets. The following definitions generalize Przymuzinsky’s concepts to our programs and models. The reader should note that the definitions are stated in terms of models and facts only, there is no dependence on the fact that our structures contain sets as elements. Thus, the generalization is straightforward.

**Definition 4.4.** Let \( P \) be a layered program, \( M \) be a set of \( U \)-facts, and \( N \) and \( Q \) be distinct models of \( P \). We say that \( N \) is preferable to \( Q \) w.r.t. \( M \), denoted \( N <_M Q \), if both \( N \) and \( Q \) contain \( M \), \( N \neq Q \), and for every fact \( p(\ldots) \in N - Q \), there exists a fact \( \hat{p}(\ldots) \in Q - N \) such that \( p \gg \hat{p} \). In words: for every fact in \( N - Q \), there exists a higher-priority fact in \( Q - N \). Thus \( N \), compared to \( Q \),
tends to minimize the extensions of higher-priority predicates (those that appear low in a layering), even at the expense of having to add facts of lower priorities (that appear higher in a layering). A model of \( P \) is perfect w.r.t. \( M \) if it contains \( M \) and there is no other model that is preferable to it w.r.t. \( M \).

It follows directly from the definition that if \( M \subset N \subset Q \), and both \( N \) and \( Q \) are models of \( P \), then \( N <_M Q \). Thus, a perfect model is minimal w.r.t. \( M \). The results proved later in this paper indicate that the converse is, in general, not true. Note that the definition does not imply that a perfect model for a program exists, or that it is unique. The issues of existence and uniqueness of perfect models will be settled by the results in the sequel.

An advantage of the concepts of preferability and perfect models is that they are based on the \( \Rightarrow \) relations between predicates, and are independent of the specific layering chosen for a program. Thus, claims about existence or uniqueness of a perfect model, once proved, apply to a layered program independently of any specific layering.

The following claim has been proved in [28] for a language with negation. The proof for our more general programs and models is essentially the same. It is reproduced here for convenience.

Lemma 4.3. Preferability with respect to a set of facts \( M \) is a transitive relation on models.

Proof. Assume \( N <_M Q <_M R \). We show that \( N <_M R \). Since, by Definition 4.4, \( <_M \) is irreflexive, we first show that \( N \neq R \), i.e., they are distinct. Assume that \( N = R \). Then we have \( R <_M Q <_M R \). Since \( Q <_M R \), \( R \subseteq Q \) is impossible. Similarly, \( Q \subseteq R \) is impossible. Hence, both \( R - Q \) and \( Q - R \) are nonempty. From \( Q <_M R \), we obtain for that for each fact \( p_i(\ldots) \) in \( Q - R \), there exists a fact \( p_j(\ldots) \) in \( R - Q \) such that \( p_i \Rightarrow p_j \). Now, from \( R <_M Q \) we obtain the existence of a fact \( p_k(\ldots) \) in \( Q - R \) such that \( p_k \Rightarrow p_j \), and so on. Since \( \Rightarrow^+ \) is transitive and irreflexive on the finite set of predicate symbols which appear in the program, it cannot have infinite decreasing chains—a contradiction. Thus, \( N \neq R \).

To show that \( N <_M R \), consider a fact \( p_i(\ldots) \) in \( N - R \). If \( p_i(\ldots) \) is also in \( Q \), then it is in \( Q - R \), so we can deduce the existence of a fact \( p_j(\ldots) \) in \( R - Q \) such that \( p_i \Rightarrow p_j \). If that fact is not in \( N \), we are done. If it is in \( N \), then it is in \( N - Q \), which allows us to infer the existence of yet another fact \( p_k(\ldots) \) in \( Q - N \) such that \( p_j \Rightarrow p_k \). If that fact is in \( R \), then it is in \( R - N \) and we are done. Otherwise, it is in \( Q - R \), so we have the same situation as we previously had for \( p_i(\ldots) \). Since there exist no infinite decreasing chains of \( \Rightarrow \), a fact in \( R - N \) must eventually be found. The case where \( p_i(\ldots) \) is not in \( Q \) is handled similarly.

While perfect models seem to capture the notion of prioritized minimization, as implied by the structure of a layered program, there is nothing in that concept that captures the specific semantics of sets. One would like to have a notion of preference that is suited for a universe with set elements, and for programs that use grouping. Unfortunately, we have been unable to find a satisfactory answer to this problem. We present in Appendix B a discussion of this subject.
Another notion of minimality can be defined given a partial order on the predicates:

**Definition 4.5.** Let $\alpha$ be a partial order on the predicates of $P$. For $N$, a model of $P$, we say that it is the minimal model of $P$ with respect to $\alpha$ that contains $M$, if

1. it contains $M$, and
2. for every predicate $p$, the extension of $p$ in $N$ is minimal, by set containment, among all models of $P$ that contain $M$ and agree with $N$ on the extensions of each predicate $q$ such that $p \alpha q$.

The relation between the two notions of minimality is summarized in the following.

**Proposition 4.1.** Let $P$ be a layered program, and let $M$ be set of U-facts. Let $N$ be a model of $P$ that contains $M$.

1. If $N$ is perfect w.r.t. $M$, then it is a minimal model of $P$ that contains $M$.
2. $N$ is preferable w.r.t. $M$ to all other models of $P$ that contain $M$ iff it is minimal w.r.t. $\Rightarrow$ and $M$.

**Proof.** (1): This follows from the definition of a perfect model.

(2): Assume $N$ is preferable to all other models of $P$ w.r.t. $M$. Then it satisfies the first condition for being minimal w.r.t. $\Rightarrow$ and $M$. Assume it does not satisfy the second condition, and let $p$ be a predicate, of highest priority, such that the condition is not satisfied for its extension in $N$. Let $Q$ be a model that agrees with $N$ on the extensions of all predicates $q$ such that $p \Rightarrow q$, and such that the extension of $p$ in $N$ is not a subset of its extension in $Q$. Such a $Q$ exists, since $N$'s extension for $p$ is not minimal. Then there exists an occurrence of $p$ in $N - Q$, but there is no occurrence of a higher-priority fact in $Q - N$—a contradiction to the assumption that $N$ is preferable to $Q$ w.r.t. $M$. The opposite direction is proved similarly. $\Box$

**Example 4.1.** Consider the program $P$, and assume that $M$ is the empty set:

$q(1)$.
$q(2)$.
$p(\langle X \rangle) \leftarrow q(X)$.

$M_1 = \{q(1), q(2), q(3), p(\langle 1, 2, 3 \rangle)\}$ is a model for $P$. It is a minimal model—no subset of $M_1$ is a model of $P$. Consider $M_2 = \{q(1), q(2), p(\langle 1, 2 \rangle)\}$, which is also a model for $P$. It is also a minimal model. It is clear that every model of $P$ must contain the facts $q(1), q(2)$; hence $M_1$ is preferable to every other model of $P$ w.r.t. $M$; hence it is the unique perfect model of $P$ w.r.t. $M$.

**4.3. Bottom-Up Semantics of Layered Programs**

We proceed to show that if $P$ is a layered program then a model for $P$ can be constructed bottom-up, by induction on the layers. We continue to consider a fixed
universe $U$. We first consider the case of a two-layered program. For this case, we distinguish the three subcases of simple rules, rules with negation, and rules with both negation and grouping. Then we consider arbitrary layered programs. We assume throughout that all rules are well formed.

**Definition 4.6.** A rule is *simple* if it contains no grouping in its head and no negative literal in its body.

For programs containing only simple rules, the existence and uniqueness of a minimal model, which can be constructed bottom-up, is a well-known fact [4,25]. We re-prove this fact, extending it to the case where sets are allowed as elements of the universe.

Recall the definition of the application of a rule to a set of facts. The application of $r$ to $M$, denoted $r(M)$, is defined as

$$r(M) = \{ B \theta : \theta \text{ is applicable to } r \text{ w.r.t. } M \}.$$ 

**Definition 4.7.** For a finite set $R$ of simple rules (i.e., a program), the application of $R$ to $M$, denoted $R(M)$, is defined to be $\bigcup_{r \in R} r(M)$. (This is essentially $T_R$, applied to $M$, as defined in [25].) The closure of $M$ under $R$, denoted $R^*(M)$, is defined by

$$R^0(M) = M,$$

$$R^{i+1}(M) = R(R^i(M)) \cup R^i(M),$$

$$R^*(M) = \bigcup_{i=0}^{\infty} R^i(M).$$

The definition of $R^*(M)$ assumes that all rules of $R$ are applied to $M$ at each step. However, to show that the result of applying a program to a set of facts is independent of the order of application of the rules, one may want to consider more general computations, in which only a subset of $R$ is applied to $M$ in each step. Of course, for the notion of closure to make sense, we need to guarantee some degree of fairness in the application of the rules, so that as long as the rule is applicable, it will eventually be applied. This is captured in the following definition.

**Definition 4.8.** An $R$-*sequence* is an infinite sequence $\hat{R} = R_1, R_2, \ldots$ such that each $R_i$ is a nonempty subset of $R$, and such that each rule of $R$ appears in it infinitely often. For any $R$ with more than one rule, there are many $R$-sequences. For a sequence $\hat{R}$ and for a set of $U$-facts $M$, $\hat{R}$ applied to $M$ is the sequence $\hat{R}(M) = M = M_0, M_1, \ldots$, such that for each $i > 0$, $M_i = R_i(M_{i-1}) \cup M_{i-1}$. We call this sequence of sets of $U$-facts an $R$-$M$-sequence. We denote the union of the elements of the sequence by $\hat{R}^*(M)$. 

\[\text{We say that the rule } r \text{ appears in the sequence infinitely often if for each } m, \text{ there exists } n > m \text{ such that } r \in R_n.\]
Note that a sequence $\hat{R}(M)$ is monotonically increasing by definition. Its union can therefore be considered as its limit. The “limit” is the least upper bound of the sequence under set inclusion. Also note that the sequence $\{R'(M)\}_{i=0}^{\infty}$ defined above is a special case, obtained from the sequence $\{R_i = R'_i\}_{i=0}^{\infty}$.

**Proposition 4.2** (Simple). Let $R$ be a set of simple rules, let $M$ be a set of $U$-facts, let $\hat{R}$ be an $R$-sequence, and let $M_0, M_1, \ldots$ be the corresponding $R$-$M$-sequence. Then:

1. The limit $\hat{R}^*(M)$ is a model of $R$ containing $M$.
2. Every model of $R$ containing $M$ contains $\hat{R}^*(M)$.

**Proof.** (1): That $\hat{R}^*(M)$ contains $M$ follows from its definition. Assume $\hat{R}^*(M)$ is not a model. Then there is a rule $r \in R$ of the form

$\text{head} \leftarrow B_1, \ldots, B_n$,

and binding $\theta$ such that $\{B_1\theta, \ldots, B_n\theta\} \not\subseteq \hat{R}^*(M)$, but $\text{head} \theta \not\in \hat{R}^*(M)$. For each $i$, there exists a $j$ such that $B_i\theta \in M_j$. Let $\alpha(i)$ denote this $j$. Let $m = \max \{\alpha(i) : 1 \leq i \leq n\}$. Since each rule is used infinitely often, for some $m' > m$, the rule $r$ is in $R_{m'}$. Clearly, $\text{head} \theta \in R_{m'}(M_{m'-1})$, and by monotonicity $\text{head} \theta \in \hat{R}^*(M)$—a contradiction.

(2): Let $N$ be a model of $R$ containing $M$. We prove by induction that $M_i \subseteq N$. For the basis, we are given that $N$ contains $M$. Assume the claim holds for $M_i$, and consider $M_{i+1}$. Now, $R(N) \subseteq N$; since $R_{i+1} \subseteq R$ and $M_i \subseteq N$, it follows that $R_{i+1}(M_i) \subseteq N$; hence also $M_{i+1} \subseteq N$. □

**Corollary 4.2** (Simple). For $R$ and $M$ as in the proposition:

1. The limits of all $R$-$M$-sequences are equal to each other. In particular they are all equal to $\hat{R}^*(M)$.
2. This limit is the unique minimal model of $R$ containing $M$.

**Proof.** (1): Consider two $R$-$M$-sequences. By (1) of the proposition, the limit of each is a model of $R$ that contains $M$. By (2) of the proposition, each contains the other; hence they are equal.

(2): By (2) of the proposition, for any model $N$ of $R$ that contains $M$, it is the case that $N$ contains $\hat{R}^*(M)$. Hence this limit is the unique minimal model of $R$ that contains $M$. □

Note that we are using here the classical definition of minimality, based on set containment. Since the program does not use grouping or negation, the relation $\gg$ is empty, so we have that $N <_M Q$ iff $N \subseteq Q$. Thus, preferability coincides in this case with set containment, and the unique minimal model is also the unique perfect model.

We now proceed to discuss the case where the rules may contain negation, but no grouping. There is no problem in extending the definitions of $R(M)$, $R'(M)$, $\hat{R}^*(M)$, and $R$-$M$-sequences to cover sets $R$ which may obtain rules having negated body literals (but no grouping). Since facts can only be added to the extensions of predicates, each $R$-$M$-sequence is still monotonically increasing, so
its union may be considered as its limit. Note, however, that in such a sequence, when a rule with negation is applied, each negative literal is evaluated on the extension of its predicate as produced by the previous steps of the sequence. Such a literal may evaluate under a binding \( \theta \) to \textit{true} in a given step, but in a later step, it may evaluate to \textit{false} under the same \( \theta \). Therefore, when negation is used, the limit may depend on the specific sequence of subsets of \( R \), that is, it may depend on the order in which rules are applied. Distinct sequences may have different limits.

Recall that by the definition of layering, any program containing derived predicates has at least two layers. If in the program negation is applied only to base predicates, then it has a layering that contains two layers only—the first layer contains the base predicates, the second layer contains all the other predicates. It is therefore two-layered. The fact that negation is applied to base predicates only, and these do not change as rules are applied, is the key to the following results.

**Definition 4.9.** For a subset \( Q \) of the predicates of \( R \) and for a set \( M \) of \( U \)-facts, let \( \text{ext}(Q, M) \) denote the extensions of the predicates of \( Q \) in \( M \). Also, let \( B(R) \) denote the set of base predicates of \( R \). That is, \( B(R) \) is the set of predicates in the bottom layer of \( R \).

**Proposition 4.3 (Negation).** Let \( R \) be a two-layered set of rules without grouping, let \( M \) be a set of \( U \)-facts, let \( \bar{R} \) be an \( R \)-sequence, and let \( M_0, M_1, \ldots \) be the corresponding \( R \)-\( M \)-sequence. Then:

1. \( \bar{R}^*(M) \) is a model of \( R \) containing \( M \), and \( \text{ext}(B(R), M) = \text{ext}(B(R), \bar{R}^*(M)) \).
2. If \( N \) is a model of \( R \) containing \( M \) such that \( \text{ext}(B(R), N) = \text{ext}(B(R), M) \), then \( \bar{R}^*(M) \subseteq N \).

**Proof.** (1): We first prove by induction on \( i \) that \( \text{ext}(B(R), M_i) = \text{ext}(B(R), M) \). The claim is obviously true for \( i = 0 \), since \( M_0 = M \). Since the extensions of base predicates do not change when rules of \( R \) are applied to \( M_i \), the claim follows.

Next, we show that \( \bar{R}^*(M) \) is a model of \( R \). Let \( r \in R \) be

\[
B \leftarrow B_1, \ldots, B_n, \neg C_1, \ldots, \neg C_m
\]

and assume that for some \( \theta \), \( B_\theta \in \bar{R}^*(M) \) for all \( k = 1, \ldots, n \), and \( C_\theta \in \bar{R}^*(M) \) for \( j = 1, \ldots, m \). Since the sequence \( \bar{R}(M) \) is monotonically increasing, and \( \bar{R}^*(M) \) is its limit, \( C_\theta \in M_h \) for all \( h \geq 0 \). For each \( k, k = 1, \ldots, n \), there exist \( \alpha(k) \) such that \( B_\theta \in M_{\alpha(k)} \), so for a sufficiently large \( q \), each \( B_\theta \) is in \( M_q \). Since each rule is applied in \( \bar{R} \) infinitely often, for some \( l > q \), \( r \) is one of the rules applied to \( M_l \), producing \( M_{l+1} \). It follows that \( B_\theta \in M_{l+1} \), hence also \( B_\theta \in \bar{R}^*(M) \).

(2): Let \( N \) be as in the lemma. We show by induction that \( M_i \subseteq N \). For \( i = 0 \), the claim is trivial, since it is given that \( N \) contains \( M \). Assume the claim holds for \( M_i \), and consider \( M_{i+1} \). Let \( r \) be a rule in \( R_{i+1} \), of the form \((*)\) above, whose application to \( M_i \) adds \( B_\theta \) to \( M_{i+1} \). Since \( \text{ext}(B(R), N) = \text{ext}(B(R), M) = \text{ext}(B(R), M_i) \), and negation is applied in rules of \( R \) only to base predicates of \( R \), the body of the rule, under the binding \( \theta \), is true in \( N \). Now, \( N \) is closed under application of rules from \( R \), since it is a model of \( R \). It follows that \( B_\theta \in N \). Thus, \( M_{i+1} \subseteq N \).
Corollary 4.3 (Negation 1). For $R$ and $M$ as in the proposition:

1. The limits of all $R$-$M$-sequences are equal to each other. In particular they are all equal to $R^*(M)$.

2. This limit is preferable w.r.t. $M$ to every model of $R$ that contains $M$.

3. The limit is the unique minimal model among the models of $R$ that contain $M$ and agree with $M$ on the extensions of the predicates in $B(R)$.

Proof. (1): The equality of all limits to each other and to $R^*(M)$ follows from the proposition precisely as in Corollary 4.2.

(2): Let $N$ be any model of $R$ that contains $M$ that is different from $R^*(M)$. If $\text{ext}(B(R), N) = \text{ext}(B(R), M)$, then by (2) of the proposition, $R^*(M) \subseteq N$; hence $R^*(M)$ is preferable to $N$ w.r.t. $M$. If $\text{ext}(B(R), N) \neq \text{ext}(B(R), M)$, then since $N$ contains $M$, it must contain some fact on the base predicates that is not in $M$, and hence also not in $R^*(M)$. It follows that in this case also, $R^*(M)$ is preferable to $N$ w.r.t. $M$.

(3): This part follows directly from (2) of the proposition. \(\square\)

The final subordinate case of two-layered programs to consider is that of programs with grouping, where the body of a rule in which grouping is applied contains only base predicates. Thus, negation and grouping are applied only to base predicates. Recall that in the following, we assume the liberal semantics, so the problem of undefined grouping does not arise.

Note that $R(M)$, $R'(M)$, $R^*(M)$, and $R$-$M$-sequences can be defined for arbitrary (not necessarily two-layered) sets of rules $R$ with grouping. Sequences are still monotonically increasing. As for programs with negation, in each sequence, when a rule is applied, only the current extensions of the predicates are considered. Hence the limits of distinct sequences may be different.

Proposition 4.4. Let $R$ be a two-layered set of rules, possibly with grouping rules, and possibly with negative literals in bodies of some rules; let $M$ be a set of U-facts; let $R$ be an $R$-sequence; and let $M_0, M_1, \ldots$ be the corresponding $R$-$M$-sequence. Then

1. $\hat{R}^*(M)$ is a model of $R$ that contains $M$, and $\text{ext}(B(R), M) = \text{ext}(B(R), \hat{R}^*(M))$;

2. if $N$ is a model of $R$ containing $M$ such that $\text{ext}(B(R), M) = \text{ext}(B(R), N)$, then $\hat{R}^*(M) \subseteq N$.

Proof. (1): We first prove by induction on $i$ that $\text{ext}(B(R), M_i) = \text{ext}(B(R), M)$. For $i = 0$, this property is given. For the induction, as in (1) of Proposition 4.3, since the extensions of base predicates are not changed by rule applications, we have that

$$\text{ext}(B(R), M_{i+1}) = \text{ext}(B(R), M_i) = \text{ext}(B(R), M).$$

The proof that $\hat{R}^*(M)$ is closed under applications of rules that do not contain grouping is the same as in the proof for (1) of Proposition 4.3. We need now also
to show closure under grouping rules. Since a grouping rule’s body contains only base predicates, its application to any one of the $M_i$’s produces the same result. Once such a rule has been applied, additional applications of it will not produce additional facts. It follows that $\hat{R}^*(M)$ is also closed under application of the grouping rules; hence it is a model. (Notice that some of the sets generated when a grouping rule is applied may not satisfy the grouping condition. The claim is still valid, since these sets are not taken into account in testing whether the rule is satisfied in the resulting sets of facts.)

(2): Assume now that $N$ is a model of $R$ that contains $M$, such that $\text{ext}(B(R), N) = \text{ext}(B(R), M)$. We prove by induction that $M_i \subseteq N$, and that the extensions of the base predicates are the same in $M$, $N$ and in each $M_i$. The proof follows the argument for part (2) of Proposition 4.3, except that we need to consider the additional case of a grouping rule. Note that the fact that the extensions of the base predicates are the same in $M$ and in $M_i$ has been proved in part (1). That $N$ has the same extension is given. Thus, we only need to prove that $M_i \subseteq N$. For $i = 0$, this is given. Assume it holds for $M_i$, and consider $M_{i+1}$. Consider a rule that is applied to $M_i$ to produce a fact in $M_{i+1}$. If it is a nongrouping rule, then the new fact is also in $N$, as shown in the proof of Proposition 4.3. If it is a grouping rule, than we note that only base predicates appear in its body, and their extensions are the same in $N$ and in $M_i$. It follows that the new fact is in $N$. □

From the proof it follows that when $R$ is two-layered, each grouping rule of $R$ needs to be applied only once. Without loss of generality, we can apply all such rules in the first step, and then proceed to apply the other rules to obtain the limit of the sequence. Let us split $R$ into two disjoint parts $R_1$ and $R_2$ such that $R_1$ contains all the grouping rules and $R_2$ contains the remaining rules. Then we have:

**Corollary 4.4 (Grouping 1).** Let $R$ and $M$ be as in the proposition, and let $R_1$ and $R_2$ be as defined above.

(1) The limits of all $R$-$M$-sequences are equal to each other. In particular they are all equal to $R^*(M)$, and also to $R_2^*(R_1(M))$.

(2) This limit is preferable w.r.t. $M$ to every model of $R$ that contains $M$.

(3) It is the unique minimal model among all models of $R$ that contain $M$ and agree with $M$ on the extensions of the predicates in $B(R)$.

**Proof.** (1): This part is proved like part (1) of Corollary 4.3. The only addition is that we need also to prove that the common limit of the sequences is equal to $R_2^*(R_1(M))$. The argument for that was presented above.

(2): The proof of this claim is identical to the proof of (2) of Corollary 4.3.

(3): The proof of this case is identical to that of (2) of Corollary 4.3. □

Note that the proof above does not rule out the possible existence of a model $N$ of $R$ that contains $M$ and that also contains ground instances of predicates from $B(R)$ not in $M$. Such a model does not necessarily contain $R^*(M)$, and it may in
fact be a minimal model, incomparable to \( R^*(M) \). This may be the case even if \( R \)

is a positive program. However, \( R^*(M) \) is preferable w.r.t. \( M \) to any such model.

We now turn to arbitrary layered programs. Let \( P \) be a layered program, and let \( L_0, \ldots, L_n \)

be a partition of the predicate symbols of \( P \) into layers. Let \( P_i \)

denote the set of all rules of \( P \) that contain predicates of \( L_i \) in their heads. Recall

that \( P \) may contain facts that are instances of predicates of \( L_i \); these are also in

\( P_i \). Clearly, \( P_0 \) is empty, and \( P_1, \ldots, P_n \) is a partition of \( P \). Let \( \overline{L}_i = \bigcup_{j=0}^i L_j \), and let \( \overline{P}_i = \bigcup_{j=1}^i P_j \).

Lemma 4.4. Let \( M_i \) be a set of ground instances of the predicates in \( \overline{L}_i \) that is a

model of \( \overline{P}_i \), and let \( M_{i+1} \) be a set of ground instances of the predicates of \( L_{i+1} \)

such that \( M_i \cup M_{i+1} \) is a model of \( P_{i+1} \). Then \( M_i \cup M_{i+1} \) is also a model of \( \overline{P}_{i+1} \).

Proof. Every rule of \( P_{i+1} \) is satisfied in \( M_i \cup M_{i+1} \) by assumption. The predicates

appearing in rules of \( \overline{P}_i \) all belong to \( \overline{L}_i \). It follows that a rule of \( \overline{P}_i \) is satisfied in

\( M_i \cup M_{i+1} \) if and only if it is satisfied in \( M_i \). The claim follows. \( \square \)

Theorem 4.1. Let \( P \) be a layered program, let \( \hat{L} = L_0, \ldots, L_n \) be a layering for \( P \), and

let \( M \) be a set of \( U \)-facts. Define \( M_0 = M \), and inductively on \( i \), let \( M_i = P_i^*(M_{i-1}) \).

Then:

(1) \( M_n \) is a model of \( P \) that contains \( M \), and for each \( i, 0 \leq i < n \), it agrees with

\( M_i \) on the extension of the predicates of \( \overline{L}_i \).

(2) \( M_n \) is preferable w.r.t. \( M \) to every other model of \( P \) that contains \( M \). Hence it

is the unique perfect model of \( P \) w.r.t. \( M \). It is also minimal w.r.t. \( \triangleright \) and \( M \).

Proof. (1): We show, using induction on \( i \), that \( M_i \) is a model of \( \overline{P}_i \) that contains

\( M \), and for all \( j, 0 \leq j < i \), it agrees with \( M_j \) on the predicates of \( \overline{L}_j \). When \( i = n \),

we obtain statement (1) of the theorem.

For the basis, \( i = 0 \), \( M_0 \) is trivially a model of \( \overline{P}_0 \), and there is no \( j, 0 \leq j < i \), so

the claim is true. Assume the claim holds for some \( i \geq 0 \). By Proposition 4.4 and

Corollary 4.4, \( M_{i+1} \) is a model of \( P_{i+1} \) that agrees with \( M_i \) on the predicates of \( \overline{L}_i \).

By Lemma 4.4, \( M_{i+1} \) is also a model of \( \overline{P}_{i+1} \). Since it agrees with \( M_i \) on the

predicates of \( \overline{L}_i \), and since by the induction hypothesis \( M_i \) agrees with \( M_j \) on the

predicates of \( \overline{L}_j \) for all \( j < i \), it follows that \( M_{i+1} \) agrees with \( M_j \) on the predicates of \( \overline{L}_j \) for all \( j < i + 1 \).

(2): Let \( N \) be a model of \( P \) that contains \( M \) and is different from \( M_n \).

Obviously, the restriction of \( N \) to the extensions of the predicates in \( \overline{L}_i \) is a model of \( \overline{P}_i \) that contains \( M \). Denote this restriction by \( N_i \). Now, let \( j \) be the smallest

integer such that \( M_j \neq N_j \). Then \( M_i = N_i \) for all \( i < j \). From Proposition 4.4 and

Corollary 4.4 it follows that \( M_i \subseteq N_j \). Therefore, if \( M_n \) \( \not\subseteq \) \( \overline{L}_i \), every predicate occurrence in this difference must belong to \( L_k \), for some \( k > j \). It follows that \( M_n \)

is preferable to \( N \) w.r.t. \( M \). The rest of (2) follows from Proposition 4.1. \( \square \)

Corollary 4.5.

(1) If \( M = \emptyset \), then \( M_n \) is the (unique) perfect model of \( P \), and in particular, it is

a minimal model of \( P \).

(2) The model \( M_n \) defined in the theorem is independent of the specific layering

used in its construction.
Recall that our constructions so far were carried out in a fixed universe $U$. The model of program $P$ obtained from the set of facts $M$ in $U$ by bottom-up layer-by-layer construction is denoted by $P^*(U, M)$, and called the intended model of $P$ w.r.t. $(U, M)$.

The results above were all obtained for the liberal semantics. To what extent do they apply for the conservative semantics? A partial answer is given by the following.

**Proposition 4.5.** The following are equivalent:

1. $(U, P^*(U, M))$ is a conservative model of $P$.
2. For all grouping rules in $P$, their applications in the construction of $P^*(U, M)$, as described above, is well defined under the conservative semantics.
3. For each $i \leq n$, $(U, M_i)$ is a conservative model of $P_i$.

**Proof.** (1) $\Rightarrow$ (2): Each grouping rule belongs to a layer $i + 1$ for some $i < n$. The predicates in its body are in $L_i$, and their extensions are in $M_i$. In particular, the extensions of these predicates in $M_i$ and in $M_n = P^*(U, M)$ are equal. Since $M_n$ is assumed to be a conservative model, the application of each grouping rule on it is well defined. The claim follows.

(2) $\Rightarrow$ (3): The claim is proved by induction on $i$. It holds vacuously for $i = 0$, since $P_0$ is empty. Assume it holds for $i$, and consider the construction of $M_{i+1}$. The applications of the grouping rules in this step are well defined by assumption. The claim follows now from Lemma 4.4 and the fact that the extensions of all predicates that appear in grouping rules of $P_{i+1}$ are the same in all $M_j$, $j \geq i + 1$.

(3) $\Rightarrow$ (1): Obvious, since (1) is a special case of (3).

Finally, we examine the relationship of stratification and set enumeration. If rules contain enumerated terms in the head, then all of these can be moved to the body of the rule. Then all the instances of enumerated sets can be expressed using $\text{scons}$. As already seen, $\text{scons}$ can be simulated using set membership and grouping. Furthermore, the fragment of code simulating $\text{scons}$ is stratified, and its addition to any stratified program (which does not define $\text{scons}$ and does not use the same “auxiliary” predicate symbols) preserves stratification. A similar argument holds for the union operator (i.e., expressions of the form $S \cup T$ which may appear in the head or the body). Nevertheless, the discussion above clearly demonstrates that it is often much easier for a programmer to write a layered program using these “auxiliary” predicates, than to write a layered program using only membership and grouping. A practical language should therefore allow their use, including their use in rule heads.

**5. CONDITIONS FOR BOUNDED GROUPING**

The results of the last section were proved for the liberal semantics. That is, in the construction of the intended model for a program, partial application of grouping was allowed. In this section we investigate conditions on programs that guarantee that whenever a grouping rule is applied in the bottom-up construction of the intended model, all the sets that are generated are in $U_\alpha$, for a large enough $\alpha$. 
(The "size" of $\alpha$ depends of course on both $P$ and $M$.) The satisfaction of such a condition by a program implies that its meaning is the same whether the conservative or liberal semantics is used, and in particular, its intended model w.r.t. $(U_\alpha, M)$ is a conservative model.

Recall the notation of Section 3, concerning a grouping rule and the bindings for it. We say that a rule $r$ is bounded for grouping on $M$ in $U_\alpha$, abbreviated as $g$-bounded, if for all $j$, $p\Sigma_j$ is a $U_\alpha$-fact. Our goal is to identify conditions on programs that guarantee $g$-boundedness of each rule w.r.t. to the set of facts to which it is applied in the construction of the intended model. Most desirable are simple conditions that can be checked at compile time and that do not overrestrict the class of allowed programs.\footnote{Note that grouping conditions such as requiring that $p\Sigma_j$ be finite, or that it be in $U_\alpha$ (or in $U_\alpha$ for any fixed, infinite $\alpha$), cannot be checked at compile time or even at run time. Indeed, it is known that for a Datalog program and a query, the problem of whether the result is finite is undecidable. This can be used to show that checking whether a program may generate an infinite set by grouping is undecidable. A similar claim holds for the question whether the answer is in $U_\alpha$.} Two conditions that are decidable and that can be effectively checked at compile time are presented below.

The following observation on the construction of the intended model is useful. Assume that $P$ is two-layered, and that $M$ is a set of $U$-facts. By Corollary 4.4, all grouping rules of $P$ are applied to $M$ in the first step, and need not be applied again; thus we have the lemma below.

**Lemma 5.1.** Let $P$ be a two-layered program, let $M$ be a set of $U$-facts, and assume that all grouping rules of $P$ are $g$-bounded on $M$ in $U$. Then $P^*(M)$ is a conservative model of $P$.

5.1. Covering

The basic idea here is that we need a restriction that guarantees that a rule applied to a "small" collection of facts adds to it only a "small" collection of "small" facts. The problem of adding "too many" facts is closely related to that of domain independence [34], also called safety, for nonrecursive queries [37].\footnote{Safety has also been used in a different sense, meaning a finite answer, in [37].}

Intuitively, domain independence means that the result of a query is not changed if the underlying domain is enlarged. (Note that if a program allows a "large" extension for a predicate to be created, then a grouping rule with an occurrence of the predicate in its body will generate a "large" fact.)

Consider a rule $r$ with a negative literal $\neg q_1(X)$ in its body. Given a set of facts $M$, we consider bindings $\theta$ for $X$ such that $q_1(X\theta)$ is in the complement in the universe of $\text{ext}(q_1, M)$. This complement is usually large, and if $X$ appears in the head of the rule, we may generate either a "too large" set of facts (when $X$ is not grouped upon), or a fact which is "too large" (when $X$ is grouped upon). This is not the case, however, if $X$ also appears in the body in a positive literal, say $q_2(X)$. Then, the space of bindings for $X$ is not the complement of $\text{ext}(q_1, M)$, but rather the difference $\text{ext}(q_2, M) - \text{ext}(q_1, M)$, which for a "reasonable" $M$ is "small". Another way a "too large" set of facts may be generated is when a variable appears in the head of the rule, but not in its body. The space of bindings
for such a variable is \( U \), and it is independent of the bindings for the other variables. Finally, if a variable appears in the head, and it appears in the body only in instances of built-in predicates, such as \( \text{mem} \), then the space for bindings for it may be too large; although we have considered built-in predicates in the definition of layering to be base predicates, their extensions are too large to be grouped upon.

Let us consider first rules without the built-in predicates \( = \) and \( \text{mem} \).

**Definition 5.1.** The covered terms of a rule are defined as follows:

1. a constant is always covered;
2. a term that appears in a positive atom in the body is covered;
3. if \( f(t_1, \ldots, t_n) \) is covered, then so is \( t_i \), \( 1 \leq i \leq n \);
4. if \( t \) is \( f(t_1, \ldots, t_n) \), and \( t_1, \ldots, t_n \) are covered, then so is \( t \).

Next, consider the built-in predicates. An occurrence of a variable in a positive instance of a predicate in the body restricts the relevant bindings for this variable to the elements that appear in the extension of that predicate in the given set of facts. This claim fails, however, for built-in predicates, since their extensions are fixed, and each element of the universe appears in an instance of each of \( = \) and \( \text{mem} \). Grouping on a variable that is restricted only by an occurrence of that variable in one of them does not restrict the possible bindings for that variable in any meaningful way, unless some of the other variables in the same built-in predicate are already known to be covered.

**Definition 5.1 (Continued).** If the rule contains instances of built-in predicates, then the following conditions are also used to determine covered terms:

4. if \( t_1 \neq t_2 \) appears in the body, then if \( t_1 \) is covered, so is \( t_2 \) (and vice versa);\(^\text{13}\)
5. if \( \text{mem}(t, S) \) appears in the body, and \( S \) is covered, so is \( t \).

Note that to show that a term is covered in a rule we have either to show that it occurs in a positive literal, or to connect it through one of the cases (2)-(5) to other terms that have previously been shown to be covered. A proof that a term \( t \) is covered is a sequence of terms such that \( t \) is the last term, and such that each term in the sequence can be shown to be covered by case (1), or by connecting it by one of the cases (2)-(5) to terms that occur before it in the sequence.

**Definition 5.2.** A rule satisfies the weak covering condition if (1) each term in its head is covered, and (2) if the head contains a grouping expression \( \langle X \rangle \), then \( X \) is covered. A rule satisfies the covering condition if in addition to the above, every term that occurs in a negative literal in its body is covered.

\(^\text{13}\)We could extend this definition further, e.g., \( f(X, t_1) = f(t_2, Y) \) is covered if both \( t_1 \) and \( t_2 \) are covered. However, we leave a more extensive formulation for future work.
Lemma 5.2. Let $R$ be a two-layered program whose rules satisfy the weak covering condition, and let $M$ be a set of $U^{\beta}_p$-facts. Then $R^*(M)$ is a set of $U^{\beta+1}_p$-facts. Hence, if $U = U^{\alpha}_a$ for $\alpha \geq \beta + 1$, then $R^*(M)$ is a conservative model of $R$.

Proof. Consider a binding that maps the positive literals (excluding $=$ and $\text{mem}$) of the body to a set of $U^{\gamma}_p$ facts, $N$. Then all terms are mapped to elements of $U^{\gamma}_p$. It follows that every covered term of the rule is mapped to an element of $U^{\gamma}_p$. This can be proved by induction on the length of the proof that the term is covered. In particular, the terms that occur in the head are covered. Hence, they are mapped to elements of $U^{\gamma}_p$.

Now, by the results in Section 4, the grouping rules of $R$, if its contains such rules, may be applied first. Consider a grouping rule of $R$. Since the elements used in $M$ are all in $U^{\beta}_p$, the bindings to the variable $X$ that appears in the grouping term $\langle X \rangle$ all assign to it values from $U^{\beta}_p$. It follows that the grouped sets are all in $U^{\beta+1}_p$, whereas all other nongrouped terms in the head are mapped to elements of $U^{\beta}_p$. Let the result of applying the grouping rules be $M'$. By the arguments above, it is a set of $U^{\beta+1}_p$-facts. The other rules are applied after the grouping rules. By the same arguments, they only generate $U^{\beta+1}_p$-facts. It follows that $R^*(M)$ is a set of $U^{\beta+1}_p$-facts. The second part of the claim follows by Lemma 5.1. \Box

Note that a program such as in Example 3.3 is ruled out by this lemma, as it creates sets of arbitrary nesting depths, given only simple atoms, which is impossible by Lemma 5.2. This is due to the use of enumerated sets in Example 3.3. Indeed, consider the rules we used in Section 2 to simulate $\text{scons}$:

\[
\begin{align*}
  p(T, S, T). \\
  p(T, S, X) \leftarrow & &\text{mem}(X, S). \\
  \text{scons}(T, S, \langle X \rangle) \leftarrow & &p(T, S, X), \text{mem}(Y, S). \\
  \text{scons}(T, \emptyset, \langle X \rangle) \leftarrow & &p(T, \emptyset, X).
\end{align*}
\]

The first two rules do not satisfy the covering condition, so the covering conditions rule out the use of set enumeration as a defined predicate. That seems to be an exorbitant price to pay. This problem can be solved by introducing $\text{scons}$ explicitly into the language, as a built-in predicate. Then we can add a covering rule for it:

\[\text{(6)} \quad \text{If } t \text{ and } S_1 \text{ in } \text{scons}(t, S_1, S_2) \text{ are covered, then so is } S_2, \text{ and if } S_2 \text{ is covered, then so are } t \text{ and } S_1. \text{ (In simple words, if the enumerated set is covered, so are its elements, and if all the elements are covered, so is the set.)}\]

We can now state a result analogous to Lemma 5.2 above.

Lemma 5.3. Let $R$ be a two-layered program whose rules satisfy the weak covering condition, and let $M$ be a set of $U^{\beta}_p$-facts. If $R$ uses set enumeration, then $R^*(M)$ is a set of $U^{\beta+\omega}_p$-facts. Hence, if $U = U^{\alpha}_a$ for $\alpha \geq \beta + \omega$, then $R^*(M)$ is a conservative model of $R$.

Proof. The proof is essentially like that for the previous lemma, except that set enumeration needs to be taken into account. Consider then the built-in predicate $\text{scons}$. If $X$ is mapped to an element of $U^{\gamma}_p$, then the term $\langle X \rangle$ is mapped to an element of $U^{\gamma+1}_p$. More generally, if $X$ and $Y$ in $\text{scons}(X, Y, Z)$ are mapped to
elements of \( U_\alpha \), then \( Z \) is mapped to an element of \( U_{\alpha + 1} \). It follows that if a rule contains occurrences of \texttt{scors}, and the rule is applied to a set of \( U_\gamma \)-facts, it generates a set of \( U_{\gamma + k} \)-facts, where \( k \) is determined by the rule's body. Since such rules may be recursive (see Example 3.31), the best limit we can obtain when such rules are applied to a set of \( U_\beta \)-facts is that the result is a set of \( U_{\beta + \omega} \)-facts. The other details of the proof carry over.

We now present an argument to the effect that we can actually assume that the rules satisfy the covering condition, not just the weak covering condition. The significance of the covering condition is that it implies that the result of a program is essentially independent (see Lemma 5.5) of the extended universe in which it is computed.

**Lemma 5.4.** Let \((U, M)\) be an interpretation such that there exists an element of \( U \) that does not appear in any fact of \( M \). Let \( r \) be a rule that satisfies the weak covering condition, but does not satisfy the covering condition, and let \( r' \) be obtained from \( r \) by deleting all the negative literals that contain uncovered terms. Then \( r(M) = r'(M) \).

**Proof.** Clearly, if a binding applied to \( r \) generates a fact from \( M \), then the same binding can be applied to \( r' \) to generate the same fact. For the converse, consider a binding \( \theta \) that, applied to \( r' \), generates a fact from \( M \). We can extend \( \theta \) to a binding for the variables of \( r \), by defining it on all the variables that appear in \( r \) but not in \( r' \), to be an element that does not appear in facts of \( M \). It is easy to see that this extended binding generates the same fact for the head as \( \theta \), since the extension of \( \theta \) to the uncovered variables only affects the truth values of negative literals in the body, and the construction of the binding guarantees that these values are all true.

Now, assume that \( M \) is a set of \( U_\beta \)-facts, and that the universe is \( U_\alpha \), for some \( \alpha > \beta \). Since the construction of \( U_{\beta + 1} \) involves a powerset operation, its cardinality (and that of \( U_\beta \) also, of course) is strictly larger than that of \( U_\beta \). When a grouping rule is applied to \( M \), the number of sets that are generated is bounded by the number of bindings for the body of the rule that are applicable w.r.t. \( M \). This number is bounded by the cardinality of \( U_\beta \). It follows that \( U_{\beta + 1} \) contains elements that are not in \( r(M) \). The same reasoning applies to the set of facts generated from \( M \) by a finite number of grouping rules. We have thus the following lemma:

**Lemma 5.5.** Let \( R \) be a two-layered program that satisfies the weak covering condition, and let \((U_\alpha, M)\) be an interpretation such that \( M \) is a set of \( U_\beta \)-facts for some \( \beta + \omega \leq \alpha \). Then \( R^*(M) \) is independent of the value of \( \alpha \).

**Proof.** By the discussion above we can replace each rule in \( R \) by a rule that is covered. Thus, w.l.o.g., we can assume that the rules of \( R \) satisfy the covering condition. It is easy to see that for a rule that satisfies the covering condition and for a set of \( M \), the set of bindings that generate facts in \( r(M) \) is determined by \( M \) and is independent of the universe. The claim now follows by induction on the length of an \( R-M \)-sequence that generates \( R^*(M) \).
This lemma essentially states that (weak) covering guarantees a domain-independence property. It is indeed quite close to the concept of safety used in [37] for guaranteeing domain independence for nonrecursive queries. (Note that each rule can be viewed as a first-order query.) A program has the absolute domain-independence property if every element in a tuple, belonging to the standard model, is an element of a tuple of the (interpretation of the) base relations. Since we have a set constructor that defines sets that are not necessarily in the underlying database, we cannot guarantee absolute domain independence.

The results are summarized in the following.

Theorem 5.1. Let $P$ be a layered program, with a layering $L_0, \ldots, L_n$, of all whose rules satisfy the weak covering condition, and let $M$ be a set of $U_\alpha$-facts. Then the intended model of $P$ w.r.t. $(U_\alpha, M)$ is a conservative model of $P$, for each $U_\alpha$, such that $\alpha \geq \beta + \omega * n$, and it is independent of $\alpha$. If set enumeration is not used in $P$, then $\alpha$ can be taken to be $\geq \beta + n$.

Proof. The claim follows from the previous lemmas by induction on the number of layers. □

Of course, from a database perspective, the most interesting case is when we are given a finite database. In this case $M \subseteq U_\omega$, and since $n < \omega$, we have:

Corollary 5.1. Let $M$ be a finite set of facts such that all the sets in it are finite, and let $P$ be a layered program with weakly covered rules. Then the intended model of $P$ w.r.t. $(U_\alpha, M)$ is a conservative model of $P$ for each $\alpha \geq \omega * \omega$, and it is independent of the value of $\alpha$.

In view of Theorem 5.1 and its corollary, if $P$ is a layered program with covered rules, and $M$ is finite (or bounded), then $P$ has an intended conservative model, whose value is essentially independent of the extended universe. We call this model the intended model of $P$ w.r.t. $M$.

5.2. Typing

The next condition we consider is to allow only typed programs. Various typing disciplines are possible, and we explore one that addresses most directly the problem at hand. The one we have chosen forces grouped sets to be in $U_\alpha$ when the given set of facts is in $U_\alpha$. (Note that sets in $U_\alpha$ have bounded nesting depth, but are not necessarily finite.)

For an element $e$ in $U_\alpha$, define its nesting depth as follows.

Definition 5.3.

(i) If $e \in U_0$, then its nesting depth is 0.

(ii) If $e = f(e_1, \ldots, e_n)$, then the nesting depth of $e$ is the maximal nesting depth of any of the $e_i$'s, $1 \leq i \leq n$.

(iii) If $e$ is a nonempty set, then the nesting depth of $e$ is 1 plus the maximal nesting depth of any of its elements. If $e = \{\}$, then its nesting depth is 1.
Definition 5.4. Define a type to be an integer $n$. An element $e$ is of type $n$ if its nesting depth is at most $n$.

Assume that a type is assigned to each position of a predicate. A binding is considered to be $t$-applicable to a literal if it assigns to each argument a value of the proper type for the position of that argument in the predicate. For a literal with a grouping expression, we require the binding to be $t$-applicable to each term of the literal, but there is no requirement on the grouping expression. (A binding does not assign a value to such an expression.) A binding is $t$-applicable to a rule if it is $t$-applicable to each one of its literals, including the head. Notice that a binding may be $t$-applicable to all the literals of a rule’s body, yet not be $t$-applicable to the rule, unless it is also $t$-applicable to the head. The definition of the application of a rule without grouping to a set of facts is the same as before, except that the bindings that are considered are now restricted to $t$-applicable bindings. Similarly, in the definition of truth value for a rule, only $t$-applicable bindings are considered. For a grouping rule, we change the definitions of the conservative and liberal semantics, as follows.

Definition 5.5. For a rule $r$ with head $p(\bar{T}, \langle X \rangle)$, and body $body(\bar{Z}, X)$, and for a set of $U_\omega$-facts $M$, let $\Sigma, \theta, \Sigma_j$ be as in Section 3, except that only $t$-applicable bindings are considered. Define

$$p\Sigma_j = \begin{cases} p(\bar{T}\theta, \{X\theta : \theta \in \Sigma_j\}) & \text{if } \{X\theta : \theta \in \Sigma_j\} \text{ is of the proper type for that position of } p, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Define the conservative and liberal meanings of applying $r$ to $M$ as in Section 3, taking into consideration the above modification to the definition of $p\Sigma_j$, and similarly redefine the two semantics for when the rule is true in $M$.

The effect of the new definition is that when a grouping rule is applied to a set of facts, $p\Sigma_j$ is added as a fact only if the set in the grouping position is of the proper type. (All other values are of the proper types, since only $t$-applicable bindings are used.) Since a set of any of the types is of bounded nesting depth, it is in $U_\alpha$ for all $\alpha \geq \omega$; hence for all such $\alpha$, the result of an application of a grouping rule is a $U_\alpha$-fact.

A problem that the definitions above raise is that we have changed the definitions of rule application and rule satisfaction for both grouping and non-grouping rules, so it seems we need to reexamine all our previous results. Also, it seems we have now a worse situation than before regarding grouping, since there are more opportunities for grouping to be undefined.

We offer two approaches to solve this dilemma. The first is to leave the definitions above intact. Programs now have an implicit typed semantics, since the syntax of the program contains no guarantee that typing constraints are obeyed; rather, they are enforced by the evaluation procedure. The questions above are addressed by the “use” of implicit-type predicates, as follows.

For each $n$, let $n'$ be a built-in predicate such that $n'(e)$ holds iff $e$ is of type $n$. Given a program $P$, we add to the body of each rule type literals for all the terms
that appear in it (including the head literal), according to the positions they occupy
in the literals of the rules. Thus, if \((-)p(t_1, \ldots, t_k)\) is a literal, we add to the body
the atoms \(n_i(t_1), \ldots, n_i(t_k)\), where \(n_i\) is the type of \(p\)'s \(i\)th argument. For a head
of the form \(p(\overline{T}, X)\), if the type of the position of \(X\) is \(n_i\), we attach the type
predicate \(k'(X)\), where \(k = n - 1\). For the evaluation of the modified program, we
consider the extensions of the type predicates to be part of each database, that is, part of each set of facts \(M\). If the modified program is \(P'\), and the set of facts obtained from \(M\) by adding type facts is \(M'\), then we have that the evaluation of
\(P\) on \(M\) is identical to the evaluation of \(P'\) on \(M'\). However, the rules of \(P'\)
satisfy the covering condition. Therefore, all our previous results are valid. Note, in
particular, that the extensions of the type predicates are fixed and independent
of the value of \(\alpha\) of the universe, and can therefore be considered "small". (In
contrast, the extensions of \(=\), \(\text{mem}\) depend on \(\alpha\).) Therefore, our claims that the
semantics is independent of the value of \(\alpha\), provided that \(\alpha \geq \omega\), are also valid.

The second approach is to make the type discipline explicit, visible in the syntax
of the program, and enforceable at compile time. We present one possible solution
in this direction.

**Definition 5.6.** A nongrouping rule is **type preserving** if for each binding that is
t-applicable to its body, the terms that are generated for the head are of the
proper types according to their positions in the head literal. A grouping rule,
with head \(p(\overline{T}, X)\), is **type preserving** if, in addition to the condition above,
for each set \(\Sigma_j\) of bindings that are t-applicable to its body and assign the same
values to \(\overline{T}\), the set of values assigned to \(X\) is of the proper type for the position
of \(X\) in \(p\). A program is **locally type preserving** if each of its rules is type
preserving.

**Definition 5.7.** A ground fact \(p(e_1, \ldots, e_n)\) is **properly typed** if each \(e_i\) is of the
proper type for the \(i\)th position of \(p\). A set of U-facts is **properly typed** if each
fact in it is properly typed.

It follows from our definition of a type that if \(M\) is properly typed, then all facts
in \(M\) have a bounded nesting depth, where the bound is the maximum of all types
of arguments in predicates. In particular, \(M \subseteq U_n\), where \(n\) is that maximum.

**Proposition 5.1.** Let \(P\) be a program that is locally type preserving, let \(M\) be a properly
typed set of U-facts, and let \((P^*)^{k<\alpha}\), for some ordinal \(\alpha\), be a (possibly
transfinite) sequence of subsets of \(P\). Define \(M^0 = M\), \(M^{i+1} = M^i \cup P^{i+1}(M^i)\),
where in the application of rules no type constraints are imposed on the bindings,
and for a limit ordinal \(\beta\), \(\beta \leq \alpha\), define \(M^\beta = \bigcup_{i<\beta} M_i\). Then each fact in \(M^\alpha\) is
properly typed.

**Proof.** The proof is by induction on the ordinals. It is clearly true for \(i = 0\).
Assume it holds for \(M^i\). Thus, for any rule applied to \(M^i\), all applicable bindings
w.r.t. \(M\) are t-applicable to the body of the rule. Since the rules are type
preserving, such bindings produce only properly typed facts. It follows that all the
new facts that are generated are properly typed. For a limit ordinal, the claim is
obvious, since the union of properly typed sets is properly typed. □
Proposition 5.1 implies that for any $R$-sequence from $M$, whenever a grouping rule is applied, the sets that are generated are well defined according to the revised definition of $pC_j$. Furthermore, as seen in the proof, all new facts that are generated are properly typed, so whenever a rule is applied, the set of $t$-applicable bindings is the same as the set of all bindings applicable w.r.t. the relevant set of facts. It follows immediately that all the results of Section 4 are valid if $M$ is properly typed and $P$ is locally type preserving.

**Theorem 5.2.** Let $P$ be a layered, locally type-preserving program, and let $M$ be a properly typed set of facts. Let $\alpha \geq \omega$. Then the intended model of $P$ w.r.t. $(U, M)$ is a conservative model of $P$. Its value is independent of $\alpha$.

**Proof.** Let $L_0, \ldots, L_n$ be a layering for $P$, and consider $M_i$, $i = 0, \ldots, n$. Now $M_1$ is obtained as the limit of a $P_1$-$M_0$-sequence, and generally, $M_{i+1}$ is obtained from $M_i$ by a $P_{i+1}$-$M_i$-sequence. The concatenation of these sequences is a sequence $\{P^\infty\}_{i < n \cdot \omega}$. Then it is easy to see that $M_i = M^{i \cdot \omega}$, using the notation of Proposition 5.1. By the proposition, all the facts in $M_n = M^{n \cdot \omega}$ are properly typed, so all bindings that are considered in rule applications in the construction of $M_n$ are $t$-applicable. Further, since each grouping rule is type preserving, the grouped sets are well defined according to the revised definition (Definition 5.5) whenever a grouping rule is applied in the construction of $M_n$. Thus $M_n$ is a conservative model. The independence of $\alpha$ follows from the result of the previous section. \(\Box\)

The final issue that we need to consider now is how difficult it is to check whether a program is locally type preserving.

**Proposition 5.2.** It is decidable whether a given program is locally type preserving.

**Proof:** See Appendix A. \(\Box\)

5.3. **Discussion**

In this section we have complemented the results of Section 4. The combined results of the two sections provide sufficient conditions for programs to have well-defined (i.e., conservative) semantics, which is determined by the program and by the set of facts to which it is applied, but is not dependent on the universe in which the application is assumed to take place. The restriction used to prevent the possibility of paradoxes—the covering condition—is well known from set theory, and has also received a lot of attention in the database literature. Our results reaffirm its significance. It is worthwhile to note that typing can be viewed as a special case of covering.

Covering may be too strong a restriction. It does not allow nonground facts. Also, in logic programming it is common practice to use program fragments that do not satisfy the covering condition. For example, consider the program segment at the end of Section 2 that defines `scons`. As another example, consider a program that checks if an element appears in a list. Typically, both the element and the list are variables, so the covering condition is not satisfied. However, such a program is typically invoked with both the element and the list instantiated, in which case we
feel that the spirit, if not the formal definition, of covering holds. Thus, it is
desirable to replace the covering condition by a *dynamic covering condition*, where
the covering condition is tested for given bindings in a query. Such testing involves
a propagation of bindings from rule to rule, as described in [10, 38]. This subject is
outside the scope of this paper.

Since typing is essentially a special case of covering, why should we introduce it
as a separate solution? First, although the fact that typing solves the kind of
semantic problems encountered here is known, it is worthwhile to explain how this
solution is achieved, and especially the properties of the type discipline that
support the solution. We note that data models that support sets are very popular
now, especially with the development of object-oriented database models and
systems [1, 6, 16, 19]. Many of these models support typing, sometimes implicitly.
For example, in object-oriented models, there is a distinction between object
identities and regular values. It is important to make explicit the properties of the
type disciplines of these models that prevent the languages from being too
powerful.

Second, we note that our type discipline provides a benefit in terms of how set
enumeration is used. When covering was discussed, we had two options regarding
enumeration: either not to use it (clearly an unacceptable option in practice), thus
guaranteeing “convergence” to the intended model with a finite increase in set
nesting depths, or to use it and allow programs, such as in Example 3.3, that
“converge” only at $\omega \ast \omega$. The type discipline allows the use of enumeration, yet
forbids such programs, which seems like a reasonable restriction in practice. In
particular, it removes one potential cause of infinite answers.

6. ON THE POWER OF LANGUAGE FEATURES

In this section we briefly and informally consider the power of various language
features. We show that more general grouping terms can be used in the language,
without increasing its expressive power. We compare our language with the LPS
language [22] proposed recently. Finally, we investigate the relationships between
different operations.

6.1. Extended Grouping

In this section we briefly consider syntactic extensions. We show that these
extensions do not increase the expressive power of the language, although they
definitely contribute to the ease of writing programs. The presentation here is
informal.

Recall that we did not allow terms that contain grouping expressions as
subterms, and that only one grouping expression in the head of a rule was allowed.
We now remove both of these restrictions. In the recursive definition of terms, a
grouping expression now has the same status as any other term, and a term may
contain subterms that are grouping expressions. We also allow any number of such
terms to appear in a rule’s head. Grouping is still not allowed in the body. In the
following, we refer to terms that do not contain grouping as *simple* terms, and to
terms that contain grouping in any form as *grouping* terms. We use $T, \bar{T}$ to denote a simple term or a vector of simple terms respectively, and $G, \bar{G}$ for a grouping term and a vector of grouping terms respectively. The general form for a rule head is then $p(T, G)$.

**Example 6.1.** The following are examples of grouping terms:

- $(X, f(X, \langle Y \rangle), f(X, g(Y, \langle Z \rangle, \langle W \rangle))$, $\langle g(X, Y) \rangle$.

The following are examples of rule heads:

- $p(\langle X \rangle)$, $p(X, \langle Y \rangle, \langle Z \rangle)$, $p(X, \langle h(Y, \langle Z \rangle) \rangle, Y, \langle W \rangle)$.

We now explain how each rule written in the extended syntax described above can be rewritten into a set of rules in which only the more restricted syntax, as presented in Section 2, is used. It will be seen that the extended constructs can be assigned meaning in at least two ways. To each of those there corresponds a different rewriting procedure.

We first consider rules that have more than one grouping terms in their heads. The intended meaning is that each of the grouping terms is to be evaluated independently of the others.

**Example 6.2.** Consider a relation $r(T, S, C, D)$, in which $T$ stands for Teacher, $S$ stands for Student, $C$ stands for Class, and $D$ stands for Day. The meaning of a tuple is: Teacher teaches Student in class Class on day Day. In the rule

- $p(T, \langle S \rangle, \langle D \rangle) \leftarrow r(T, S, C, D)$,

each tuple of $p$ has a teacher, the set of students taking some class with this teacher, and the set of days on which this teacher teaches some class.

Assume that a rule has the form

- $p(\bar{T}, G_1, G_2, \ldots, G_n) \leftarrow \text{body}$.

That is, grouping is performed in parallel in the rule’s head. The intended meaning of parallel evaluation is captured by translating such a rule into the following program segment:

- $q_1(\bar{T}, G_1) \leftarrow \text{body}$.
- $q_2(\bar{T}, G_2) \leftarrow \text{body}$.
- $\vdots$
- $q_n(\bar{T}, G_n) \leftarrow \text{body}$.
- $p(\bar{T}, S_1, \ldots, S_n) \leftarrow q_1(\bar{T}, S_1), \ldots, q_n(\bar{T}, S_n)$.

Next we consider a rule that has a single grouping term in its head. We first consider the meaning that can be assigned to such a rule.
**Example 6.2 (Continued).** Consider the following rule:

\[ p(T, \langle h(S, \langle D \rangle) \rangle) \leftarrow r(T, S, C, D). \]

Each tuple of \( p \) has a teacher in the first component and a set of terms \( h(t_1, t_2) \) in the second. In such a term, \( t_1 \) is a student who takes some class from the teacher and \( t_2 \) is a set of days in which the student takes some class. Two meanings are possible here. The first is that the classes are not necessarily connected to the teacher. The second is that only the classes the student takes with the teacher are included.

Now consider the rule

\[ p(T, S, \langle h(C, \langle D \rangle) \rangle) \leftarrow r(T, S, C, D). \]

In a tuple of \( p \), with each teacher-student combination there is associated a set of terms \( h(t_1, t_2) \), where \( t_1 \) is a class taken by the student and taught by the teacher, and \( t_2 \) is a set of days this class is taught. Here again, we may consider days when the class is taught by anybody, or only days the class is taught by that teacher and taken by that student.

In the following we outline how to rewrite a rule with complex grouping terms in its head into a set of rules that use only simple grouping, as defined in Section 2. As the example demonstrates, it is possible in general to assign at least two different semantic interpretations to such rules. We present the transformation for the second interpretation, namely that all the variables collected in the terms are related to each other in the body. For the rule with head \( p(T, \langle h(S, \langle D \rangle) \rangle) \) in the example above, this means that for each teacher we collect students that take classes from that teacher, and for each student the days in which he takes classes from that teacher, then we apply \( h \) to this collection, and collect again. We illustrate the procedure using this rule as an example. The idea is to perform grouping in stages, from the inside out.

**Example 6.3.** The rule

\[ p(T, \langle h(S, \langle D \rangle) \rangle) \leftarrow r(T, S, C, D). \]

is translated into the following sequence of rules:

\[ q_1(T, S, \langle D \rangle) \leftarrow r(T, S, C, D). \]

\[ q_2(T, hff(S, Y)) \leftarrow q_1(T, S, Y). \]

\[ p(T, \langle H \rangle) \leftarrow q_2(T, H). \]

### 6.2. Comparison with LPS

Kuper [22] has recently proposed a useful extension to logic programming, denoted LPS (logic programming with sets), by allowing rules of the form

\[ \text{head}(\overline{Y}, X_1, \ldots, X_n) \leftarrow (\forall x_1 \in X_1) \ldots (\forall x_n \in X_n)[B_1, \ldots, B_m]. \quad (*) \]

where \( \text{head} \) and the \( B_i \)'s are literal, the \( x_i \)'s are variables ranging over atomic elements, and the \( X_i \)'s are variables ranging over finite sets. The \( \overline{Y} \) represents other set or simple variables that appear in the body. The universe for LPS is obtained from the Herbrand universe by adding to it its finite subsets. It is
therefore strictly contained in $U_n$.\textsuperscript{14} Note that the $x_i$'s are bound in the body of the rule; they do not appear in the head. Other simple or set variables may appear in the body and the head. See [22] for precise definitions.

**Example 6.4.** Predicate $\text{disj}$ tests whether $X$ and $Y$ are disjoint sets. Predicate $\text{subset}$ tests whether $X$ is a subset of $Y$:\textsuperscript{15}

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

We show, informally, how a rule of the form (\textast) above can be simulated by a program fragment in our language. The idea is that universal quantifiers can be simulated by set comparisons—a trick that is widely used in commercial relational-database languages. The simulation contains the following (here all lowercase letters also denote variables):

\[
a(Y,X,\ldots,X_n,g(x_1,\ldots,x_n)) \leftarrow B_1,\ldots,B_n, x_1 \in X_1,\ldots,x_n \in X_n.
\]

This rule collects all tuples of $x_i$'s, where each $x_i$ is in $X_i$, such that the body predicates hold for them:

\[
b(Y,X,\ldots,X_n,\langle S \rangle) \leftarrow a(Y,X,\ldots,X_n,S).
\]

The $g$-tuples are collected into a set, for each vector of $Y$, $X_1,\ldots,X_n$ values. Clearly, we need now to select only the tuples in $b$ in which this set is the cross product of the $X_i$'s:

\[
c(X_1,\ldots,X_n,g(x_1,\ldots,x_n)) \leftarrow x_1 \in X_1,\ldots,x_n \in X_n.
\]

This rule collects tuples in the cross product of the $X_i$'s:

\[
d(X_1,\ldots,X_n,\langle S \rangle) \leftarrow c(X_1,\ldots,X_n,S).
\]

The last element in a $d$-tuple is the cross product of the first $n$ elements:

\[
\text{head} \leftarrow b(Y, X_1,\ldots,X_n, S), d(X_1,\ldots,X_n, S).
\]

The above is only a sketch. We have not handled the case where some $X_i$'s may be empty; we claim this to be a straightforward task.

If the LPS program contains recursion, then the construction above is still syntactically valid, but the resulting program is not layered. The definition of the predicate $a$ uses grouping, so the simulation of the LPS rule uses grouping. If the head predicate appears also in the body, or, more generally, any predicate that appears in the body depends through other rules on the head predicate, then we have a cycle in which at least one edge is labeled with $>$ (using the notation of Section 3). We note that, intuitively, a bottom-up construction applied to such a program generates a unique model, which may be considered as the intended model for the program. The reason for that is that grouping is used here to simulate bounded universal quantifiers. Therefore, the cycle involving $>$ on the

\textsuperscript{14}Although operators like union can be defined in LPS, they all generate finite sets from finite sets, so there is no need to consider a universe that also contains infinite sets.

\textsuperscript{15}We use $x \in X$ for $\text{mem}(x,X)$. 
predicates does not lead to ill-defined results. We do not know if a bounded universal quantifier may be simulated by grouping so that the resulting program is always layered, but we conjecture that it cannot. The conclusion is that some nonlayered programs may be considered to have well-defined semantics. That is, layering is a sufficient condition that allows us to endow a program with well-defined semantics, but more general conditions may exist. This is a subject for future research. (See [31] for some more work in this direction.)

As explained above, the universe of LPS is strictly contained in \( U_w \). It follows that there exist programs in our language that cannot be expressed in LPS. Consider the following program \( P \):

\[
p(\langle X \rangle) \leftarrow q(X).
\]
\[
\omega(\langle X \rangle) \leftarrow p(X).
\]
\[
q(1).
\]

It has the model \( M = \{ q(1), p(\{1\}), \omega(\{1\}) \} \). There is no LPS program that defines this model, since LPS programs cannot create doubly nested sets. Similarly, one may write a Datalog (with function symbols) program that defines a predicate \( q \), such that the extension of \( q \) is finite or infinite, depending on the contents of a database. If now we add a rule

\[
p(\langle X \rangle) \leftarrow q(X).
\]

we obtain a layered program, so it has a conservative intended model. If the Datalog program generates an infinite extension for \( q \), then the extension of \( p \) contains a single fact, with an argument that is an infinite set. If the extension for \( q \) is finite, then \( p \) holds on a finite set. In LPS, the Datalog portion is legal. If the extension of \( p \) can be defined in LPS, then it always involves only finite sets. Kuper proves [23] that if we restrict attention to programs that generate only singly nested sets, then grouping cannot be defined.

6.3. On the Relative Power of Language Constructs

As seen in the introduction, \( \text{mem} \) can be defined by using \( \text{scons} \) and \( \text{union} \). We have also seen how to define \( \text{scons} \), and similarly \( \text{union} \), using membership and grouping. We consider here additional relationships between language constructs.

Next, we show that grouping can be used to "simulate" negation, provided we assume the existence of a distinguished constant that is used for this purpose only. Using grouping, a negative predicate may be converted into a positive one as follows. Let \( \bot \) be a constant in the universe \( U \) whose usage is prohibited in programs (except in the following construction). Suppose that we have a rule of the form

\[
\text{head} \leftarrow B_1, \ldots, B_n, \ldots, \neg p(\overline{T}), \ldots.
\]

where the \( B_i \)'s are the positive literals of the body, and \( \neg p \) is a negative literal (there may be other negative body literals). Let \( \bar{c}, \bar{X} \) be the constants and the variables that appear in \( \overline{T} \). We assume that \( p \)'s predicate symbol is neither \( \text{mem} \) nor \( =; \) basically, because of their standard interpretation, \( \neg \text{mem} \) and \( \neg = \) can be
regarded as built-in predicates. Let \( \text{const} \) denote an arbitrary constant different from \( \bot \). We first add a rule

\[
s_1(\bar{c}, \bar{X}) \leftarrow p(\bar{T}).
\]

The occurrence of \( \neg p(\bar{T}) \) can now be replaced by \( \neg s_1(\bar{c}, \bar{X}). \) We now add the rules

\[
\begin{align*}
s_2(\bar{c}, \bar{X}, \bot) & \leftarrow B_1, \ldots, B_n. \\
s_2(\bar{c}, \bar{X}, \text{const}) & \leftarrow s_1(\bar{c}, \bar{X}), B_1, \ldots, B_n. \\
s_3(\bar{c}, \bar{X}, \langle Y \rangle) & \leftarrow s_2(\bar{c}, \bar{X}, Y).
\end{align*}
\]

It can be seen that the last element in a tuple of \( s_3 \) is a set that always contains \( \bot \) (if the positive part of the body is satisfiable). It contains \( \text{const} \) only if the other arguments of the \( s_3 \) tuple satisfy \( s_1 \). The occurrence of

\[
\neg s_1(\bar{c}, \bar{X})
\]

can therefore be replaced by

\[
s_3(\bar{c}, \bar{X}, S), \quad S = \{ \bot \}.
\]

Observe that (1) a layered program remains so after this transformation, (2) we state without proof that the intended model for the transformed program, restricted to predicates in the original program, is the intended model for the original program, and (3) if the original rule satisfied the covering condition, then the new rules also satisfy the covering condition.

We now show that if we restrict grouping to the case where the grouped set is finite, then grouping can be expressed in terms of the other constructs. Assume we have a rule of the form

\[
p(\bar{X}, \langle Y \rangle) \leftarrow \text{body}(\bar{X}, \bar{Z}, Y).
\]

We replace such a rule by the following set of rules:

\[
\begin{align*}
p_1(\bar{X}, \langle Y \rangle) & \leftarrow \text{body}(\bar{X}, \bar{Z}, Y). \\
p_1(\bar{X}, S) & \leftarrow p_1(\bar{X}, S), p_1(\bar{X}, S_2), \text{union}(S_1, S_2, S). \\
p_2(\bar{X}, S_1, S_2) & \leftarrow p_1(\bar{X}, S_1), p_1(\bar{X}, S_2), \text{union}(S_1, S_2). \\
p_3(\bar{X}, S_1) & \leftarrow p_2(\bar{X}, S_1, S_2). \\
p_4(\bar{X}, S) & \leftarrow \neg p_3(\bar{X}, S), p_1(\bar{X}, S).
\end{align*}
\]

In \( p_1 \) we collect, for each \( \bar{X} \), finite sets \( S \) such that for each \( Y \in S \) the body holds for \( \bar{X}, \bar{Z}, Y \) for some \( \bar{Z} \). In \( p_2 \) we collect with each \( \bar{X} \) two such sets, \( S_1, S_2 \), such that \( S_1 \) is a subset of \( S_2 \). In \( p_3 \) we project on the first two positions of \( p_3 \), thus obtaining sets that are not maximal. The last rule uses negation to select the maximal set that is associated with \( \bar{X} \). These rules therefore compute the grouped set, provided that it is finite. If it is not, the set \( S \) in \( p_4 \) is empty. Of course, testing for a given program whether a grouped set is finite or not is undecidable.
We can generalize the construction above to handle also infinite sets, as follows. Consider the rules

\[
\begin{align*}
    p5(\overline{X}, \overline{Z}, Y) & \leftarrow \text{body}(\overline{X}, \overline{Z}, Y). \\
p6(\overline{X}, \overline{Z}, S) & \leftarrow \text{mem}(Y, S), \neg p5(\overline{X}, \overline{Z}, Y). \\
p7(\overline{X}, \overline{Z}, S) & \leftarrow \neg p6(\overline{X}, \overline{Z}, S).
\end{align*}
\]

The rule for \( p6 \) associates with the variables of the body of the original rule a set that contains some value that does not satisfy that body. The rule for \( p5 \) is an auxiliary rule that allows the application of negation to \( \text{body} \).\(^{16} \) The rule for \( p6 \) identifies sets \( S \) containing an element that doesn't satisfy the body. The rule for \( p7 \) uses negation again to find sets that contain only \( Y \)-values that do satisfy the body. The rest of the construction is now as in the previous case. Note, however, that there is a price to pay for the generality: the construction yields a program that does not satisfy the covering condition; we do not know a construction that yields programs satisfying the covering condition. Also note that if \( p \) occurs in the body, then both constructions yield programs that contain negation on a cycle; i.e., the programs are not layered (stratified).

7. CONCLUSIONS

We have presented in this paper extensions to Horn-clause-based programming, namely, the ability to use negation, to use sets, and to apply set constructors. While negation has been treated in previous works, the extensions to sets are new. They significantly increase the expressive power of the logic-programming paradigm. The paper has concentrated on the most important constructs, namely set membership and the set grouping operator. The language being developed at MCC has other constructs, e.g., built-in functions, that further increase its power and flexibility.

Our goal in the design of \( \mathcal{DL} \)'s set constructs was to allow programmers to use sets, and set constructs that are commonly used in data-processing applications, and to consider conditions that will prevent programmers from writing ill-defined programs. Defining a set by its properties is indeed a common and useful operation. It is also commonly used by working mathematicians. Such set definitions are provided in \( \mathcal{DL} \) using the grouping operator. The language design was not an investigation of the foundations of mathematics, and our goals were certainly different from those of researches in set theory. However, "defining a set by its properties" is powerful enough so the paradoxes of set theory can appear. Further, the universal quantifier implicit in the grouping operator is nonmonotonic, and allows one to write programs with ill-defined semantics. Thus, we were forced to look for conditions that guarantee that programs have well-defined and intuitively acceptable semantics.

\(^{16} \)Note that if \( \text{body} \) contains several literals, then applying negation to each one does not produce the desired effect. Rather, we have to represent the body by a single literal, and then apply negation accordingly.
Ideally, we would like to deal with finite programs, finite databases, and finite computations. Unfortunately, while the first two objectives are easy to achieve, the third is not. It cannot be expected that a class of programs will be found such that membership in it can be checked at compile time, its programs generate only finite relations from finite databases, and it is sufficiently general. Hence, we have to allow programming constructs that enable the generation of infinite results, and cope with the problem by a combination of compile and run-time methods, possible leaving cases that will not be detected except by using a time-out mechanism. For this reason, we have considered the issue of assigning meaning to programs in infinite universes, allowing programs that perform infinite computations, and actually constructing models using transfinite induction.

We have identified two major problems: uniqueness of models and existence of models (paradoxes included). We have offered solutions to both problems. To guarantee uniqueness of the model representing the semantics of a program, we used the layering condition. This is similar to solutions used for programs with negation. A similar solution is needed if, say, the language allows universal quantifiers. To prevent the ability to express paradoxes, we imposed the covering conditions on rules. Our main result is that with these restrictions, programs have a well-defined natural semantics, which is independent of the universe in which the computation is carried out. We have also considered typing restrictions as a means of preventing paradoxes, and we have shown that they can be viewed as special cases of covering. This is another argument that emphasizes the significance of the concept of covering (and the related concept of domain independence).

We have not tried here to present a proof theory for our language. Although the expressive power of the language is not characterized here, it is known [1] that it is equivalent to a complex object calculus, which can express second-order and even higher-order queries [18]. There is no hope of finding a proof theory for such a powerful language.

We have restricted our attention throughout to extended universes, which are generalizations of Herbrand universes. These certainly capture the naive notion of sets, so their use is well motivated. For first-order logic, we have the theorem that, for universally quantified sentences, it suffices to check validity in the Herbrand universe, and for this reason logic programming is concerned mainly with the Herbrand universe. Can we make a similar claim here? Although we do not have a formal proof, we believe we cannot. There exist, of course, other models of set theory in which our language can be interpreted. Some of them satisfy the continuum hypothesis, and some do not. Our universes use the mathematicians' standard notion of sets, and we expect the hypothesis to be true in all of them, or false in all of them (for a large enough \( \lambda \)). A proof that it suffices to consider only the extended Herbrand universes would probably imply that if the hypothesis is true (false) in any model, than it is true (false) in our universes—an impossibility.

We conclude with a few research topics. First, we have presented here the condition of layering, following similar work on programs with negation. That work has since been extended considerably, to conditions such as local stratification [28, 29], and most recently to a comprehensive theory that connects logic programming with negation to constructive logic [11]. We would like to see a similar approach developed for logic programming with sets, using work on constructive set theory [17]. As mentioned in Section 5, the covering condition may be too restrictive. We
would like to formulate a dynamic, query-dependent notion. Another issue that needs additional attention is finding conditions that guarantee finiteness of the result. A key observation is that finiteness, efficiency, and covering are all to be determined, not for a program, but rather for a query on a program. A query supplies bindings for some arguments of predicates, thus defining a result that is a selection (or a projection) of a predicate that appears in the program. The bindings in the query may be propagated into the program, or the program may be rewritten so as to take advantage of the bindings as in [10]. All three issues need to be considered in this framework.

We have presented some comparisons of language features, but a complete treatment is outside the scope of this paper. Of particular significance is the relationship of the expressive power of a language to the data model for which it is defined. Our discussion of typing implies that it influences the expressive power of the language. We suspect that the interplay between the features of a model and of a language has a similar effect. As a case in point, in [19] an object-oriented model with a calculus-based language is described. Although sets are allowed, the language has a proof theory.

APPENDIX A. TYPING

In this appendix we prove the following.

Proposition. It is decidable whether a given program is locally type preserving.

Proof. We assume the program has no \(=\) and \(\text{mem}\) occurrences; later we show how such occurrences can be handled. It suffices to show that each rule is locally type preserving. Consider a rule \(r\), with variables \(X_1, \ldots, X_n\) appearing in \(r\)'s body; let \(p_1, \ldots, p_m\) be the predicate symbols in \(r\)'s body. For each \(p_i\), for each position, there is a type, i.e. an integer \(n_i, 1 \leq i \leq m\); let \(\max\) be the largest such type.

Note that if a binding is applicable to the rule's body, then it maps each term, appearing as an argument of a body literal, to an element that has a nesting depth bound by \(\max\). Every variable that appears in the body in an argument of a literal is assigned, by a binding, an element that has a nesting depth smaller or equal to the nesting depth of the element assigned to the argument in which it appears. It follows that if one wishes to check whether every \(t\)-applicable binding for the body generates a \(t\)-applicable binding for the head, it suffices to consider bindings that assign to terms elements whose nesting depths are bound by \(\max\). It is also clear that one does not need to consider actual bindings. Assuming we only know that each variable \(X_i\) has been assigned an element of nesting depth \(n_i\), we can compute the nesting depth of all the elements assigned to terms.

An assignment \(\alpha\), which associates a type with each variable in \(r\)'s body, is legal if it does not violate the types of positions of body predicates. A rule is locally type preserving iff for all legal assignments \(\alpha\), the typing for the head literal is obeyed. Consider all possible assignments of types \(t_i\) to \(X_i\), \(1 \leq i \leq n\), \(0 \leq t_i \leq \max\). Checking whether an assignment \(\alpha\) is legal is straightforward, since the type of a term can be determined if the types of its constituents are known. Similarly, given an assignment \(\alpha\), it is easy to check whether the head typing is obeyed. Since there
are only finitely many assignments to check, it is decidable whether \( r \) is locally type preserving. □

In case a rule contains \( \text{mem} \) or \( = \), it is rewritten to a set of rules containing no occurrences of \( = \) and possibly occurrences of \( \text{mem} \) in which the arguments are either variables or constants. For example

\[
h(X, Y) \leftarrow X = Y, p(Y)
\]

is rewritten as

\[
h(Y, Y) \leftarrow p(Y).
\]

Also,

\[
h(X, Y, Z) \leftarrow \text{mem}(X, \{Y, Z\}), p(X, Y, Z)
\]

is rewritten into the two rules

\[
h(X, Y, Z) \leftarrow X = Y, p(X, Y, Z)
\]

\[
h(X, Y, Z) \leftarrow X = Z, p(X, Y, Z).
\]

These two rules are each, in turn, rewritten. We do not define this transformation formally; rather, we rely on the reader's intuition. Once the program has been transformed, the above decision procedure can be easily applied.

APPENDIX B. PREFERENCE NOTION FOR PROGRAMS WITH SETS

We mentioned in Section 4 that while perfect models seem to capture the notion of prioritized minimization as implied by the structure of a layered program, there is nothing in that concept that captures the specific semantics of sets. We stated that we would like to have a notion of preference that is suited for a universe with set elements, and for programs that use grouping. We now propose such a concept of preference. Assume that \( r \) is a grouping rule with head predicate \( p(\langle X \rangle) \). If we apply \( r \) to two different sets of U-facts, say \( M_1, M_2 \) we may obtain two different sets, say \( S_1, S_2 \), such that \( p(S_1) \) is a consequence of \( M_1 \), and \( p(S_2) \) is a consequence of \( M_2 \). Each of these two \( p \)-facts appears in the one of the two interpretations that are constructed, but not in the other, and therefore set containment does not hold between the two interpretations. We note, however, that if \( M_1 \subseteq M_2 \) then \( S_1 \subseteq S_2 \). Our definition is based on this observation.

We say that an element \( s_1 \) of \( U \) is dominated by an element \( s_2 \), denoted \( s_1 \leq_d s_2 \), if one the following holds:

1. both are elements of \( U_0 \), and they are equal.
2. both are sets and for all \( a \in s_1 \), there exists \( b \in s_2 \) such that \( a \leq_d b \).
3. \( s_1 = f(e_1, \ldots, e_n) \), \( s_2 = f(e_1', \ldots, e_n') \), and for \( i = 1, \ldots, n \), \( e_i \leq_d e_i' \).

A U-fact \( e = p(s_1, \ldots, s_n) \) is dominated by a U-fact \( e' = p(s_1', \ldots, s_n') \), denoted \( e \leq_d e' \), if for \( i = 1, \ldots, n \) one has \( s_i \leq_d s_i' \). Similarly, for two sets of U-facts, \( M_1, M_2 \), we say \( M_1 \leq_d M_2 \) if each fact of \( M_1 \) is dominated by some fact of \( M_2 \).

Intuitively, it seems that the dominance relation provides a good way for comparing models. Indeed, if \( M_1 \subseteq M_2 \) and a grouping rule is applied to each of
the sets, generating a fact \( p(S_1) \) from \( M_1 \) and a fact \( p(S_2) \) from \( M_2 \), then it is the case that \( M_1 \cup \{p(S_1)\} \leq_d M_2 \cup \{p(S_2)\} \). However, consider the case when \( M_1 \) contains facts in which sets are used as terms, and \( M_2 \) is obtained from it by adding a fact that is dominated by a fact of \( M_1 \). For example,

\[
M_1 = \{ q(\{1,2,3\}) \}, \quad M_2 = \{ q(\{1,2,3\}), q(\{2,3\}) \}.
\]

Then we still have that \( M_1 \cup \{p(S_1)\} \leq_d M_2 \cup \{p(S_2)\} \). However, it is also the case that \( M_2 \cup \{p(S_2)\} \leq_d M_1 \cup \{p(S_1)\} \). In other words, dominance does not define a partial order on sets of facts—it is possible that \( M \leq N \leq M \), yet \( M \) and \( N \) are different.

Let \( M \) be a set of \( U \)-facts. We say that \( N \) is \( d\)-preferable to \( Q \) w.r.t. \( M \), denoted \( N <_{d,M} Q \), if both \( N \) and \( Q \) contain \( M \), they are different, and \( N - Q \leq_d Q - N \).

The intuition here is that by restricting attention to the differences between the sets, we factor out problems such as the one presented above. The relation \( <_{d,M} \) is irreflexive by definition. Unfortunately, it is not, in general, transitive. Przymužinski [28] defines a concept of local stratification, based on a binary relation defined on the collection of \( U \)-facts. He also shows that if all decreasing chains of that binary relation are finite, then local stratification has the same nice properties as stratification (i.e. layering), and in particular that it induces a transitive relation on models. The definition of the relation induced on the models is essentially the one we use here for inducing the relation \( <_{d,M} \). The finite-chain condition is satisfied for \( \leq_d \), for example, when all sets are finite. It is not true in general for a universe that allows nested infinite sets.

Despite the fact that \( <_{d,M} \) is not transitive in general, we may still use it to define a “minimality” concept. We say that a model \( N \) of a program \( P \) is \( d\)-perfect w.r.t. a set \( M \) if there exists no model \( Q \) of \( P \) such that \( Q <_{d,M} N \).

The relationships between the various notions of minimality are summarized in the following.

Proposition. Let \( P \) be a layered program, and let \( M \) be a set of \( U \)-facts. Let \( N \) be a model of \( P \) that contains \( M \). Then if \( N \) is preferable w.r.t. \( M \) to all other models of \( P \) that contain \( M \) and so, in particular, it is the unique perfect model of \( P \) w.r.t. \( M \), then it is \( d\)-perfect w.r.t. \( M \).

Proof. Assume that \( n \) is preferable w.r.t. \( M \) to all other models of \( P \) that contain \( M \), but that it is not \( d\)-perfect w.r.t. \( M \). Let \( Q \) be a model of \( P \) that contains \( M \) such that \( Q - N <_{d,M} N - Q \). Let \( p(t_1) \) be a fact in \( Q - N \) of highest priority, i.e., there is no \( \beta(\cdots) \) in \( Q - N \) such that \( p \gg \beta \). From the definition of \( d\)-preferable, there exists in \( N - Q \) a different fact \( p(t_2) \) such that \( t_1 <_d t_2 \). Since \( N \) is preferable to \( Q \) w.r.t. \( M \), there exists a fact \( \beta(\cdots) \) in \( Q - N \) such that \( p \gg \beta \)—a contradiction to the choice of \( p \).

Observe that the definition of dominance is not used at all in this proof.

The authors gratefully acknowledge many helpful and clarifying discussions with François Bancilhon, M. Magidor, and Carlo Zaniolo. We thank the referees of the Journal of Logic Programming; their valuable suggestions have significantly improved our presentation.
REFERENCES


