RELATIONLOG: A TYPED EXTENSION TO DATALOG WITH SETS AND TUPLES

MENGCHI LIU

This paper presents a novel logic programming based language for nested relational and complex value models called Relationlog. It stands in the same relationship to the nested relational and complex value models as Datalog stands to the relational model. The main novelty of the language is the introduction of powerful mechanisms, namely, partial and complete set terms, for representing and manipulating both partial and complete information on nested sets, tuples and relations. They generalize the set grouping and set enumeration mechanisms of LDL and allow the user to directly encode the open and closed world assumptions on nested sets, tuples, and relations. They allow direct inference and access to deeply embedded values in a complex value relation as if the relation is normalized, which greatly increases the ease of use of the language. As a result, the extended relational algebra operations can be represented in Relationlog directly, and more importantly, recursively in a way similar to Datalog. Like Datalog, Relationlog has a well-defined Herbrand model-theoretic semantics, which captures the intended semantics of nested sets, tuples and relations, and also a well-defined proof-theoretic semantics which coincides with its model-theoretic semantics.

1. INTRODUCTION

In the past decade, the nested relational and complex value models [2, 6, 14, 15, 18, 21, 25, 26] were developed to extend the applicability of the traditional relational model [13, 28] to more complex, non-business applications such as CAD,
image processing and text retrieval [3]. Extended relational algebra and calculus are provided for such kind of models. It has been proved that extended relational algebra without the powerset operator and safe calculus without the subset predicate are equivalent [26]. Queries expressed in either framework can be evaluated in polynomial time. It is shown in [6] that extended relational algebra with the powerset operator and safe extended relational calculus with the subset predicate are also equivalent and can simulate iteration and express transitive closure. However, they do so in a very inefficient way. Computations of transitive closure using either framework are inherently exponential space which means that they are not practical for any real database applications.

Another important direction of intense research has been in using a logic programming based language Datalog [10, 28] as a database query language. Such a language provides a natural way to express queries on a relational database. Furthermore, by allowing recursion and negation, it is more expressive than the traditional relational algebra and calculus [28].

In recent years, there have been some efforts to combine these two approaches, mainly by extending Datalog with set and tuple constructors [4, 9, 8, 11, 16, 17, 23]. Like Datalog, these extensions are more expressive than extended relational algebra without powerset and safe calculus without subset. However, they do not extend the expressive power when compared with extended relational algebra with powerset or extended relational calculus with subset. In either case, the main merit of these extensions is that their natural use of fixpoint construct allows us to express transitive closure declaratively in polynomial space and time [5, 6], which makes them expressive enough while still practical for real database applications.

However, these extensions suffer from two problems. One is syntactic and the other is semantic. Syntactically, they provide only primitive constructs to manipulate sets which do not allow direct access to deeply embedded values. As a result, it is cumbersome and ineffective to represent basic extended relational algebra operators such as nest, unnest, join, etc.; in these languages. Consider the following two relations:

\[
\begin{array}{cccc}
  p & & q \\
  a_1 & b_1 & c_1 & d_1 \\
  a_1 & b_1 & c_1 & d_2 \\
  a_1 & b_2 & c_1 & d_1 \\
  a_1 & b_2 & c_1 & d_2 \\
  a_1 & b_2 & c_2 & d_3 \\
  a_2 & b_1 & c_1 & d_1 \\
  a_2 & b_1 & c_1 & d_3 \\
  a_2 & b_3 & c_2 & d_1 \\
\end{array}
\]

Here \( p \) is a flat relation and \( q \) is a nested relation. We can obtain \( p \) by applying the unnest operator on \( q \) and obtain \( q \) by applying the nest operator on \( p \) defined in [26] three times respectively in an obvious way.

Now let us see how the above nest operation can be represented in two typical Datalog extensions: LDL [9, 20] and COL [4].

In LDL, sets are directly representable and set grouping and set enumeration mechanisms are provided to manipulate sets. Tuples are not directly supported but can be simulated with functors. In order to perform the above nesting operation in
LDL, we have to use grouping several times and introduce functors $f, g$ to construct nested tuples in intermediate relations $r_1, r_2, r_3, r_4$ as follows:

$$
\begin{align*}
r_1(A, B, C, \langle D \rangle) & \leftarrow p(A, B, C, D) \\
r_2(A, B, f(C, E)) & \leftarrow r_1(A, B, C, E) \\
r_3(A, B, \langle F \rangle) & \leftarrow r_2(A, B, F) \\
r_4(A, g(B, G)) & \leftarrow r_3(A, B, G) \\
q(A, \langle H \rangle) & \leftarrow r_4(A, H)
\end{align*}
$$

Unlike LDL which uses functor objects for tuples indirectly, COL directly supports tuples and sets. Furthermore, it interprets functors as mappings to sets and uses them for the set grouping and enumeration purpose. The following rules in COL show how to nest the relation $p$ to obtain the relation $q$ using interpreted functors $f, g, h$, and intermediate relations $r_1, r_2$.

$$
\begin{align*}
D \in f(A, B, C) & \leftarrow p(A, B, C, D) \\
r_1(A, B, C, f(A, B, C)) & \leftarrow r_1(A, B, C, E) \\
r_2(A, B, g(A, B)) & \leftarrow r_2(A, B, F) \\
q(A, h(A)) & \leftarrow r_4(A, H)
\end{align*}
$$

The second problem with the existing Datalog extensions is that their model-theoretic semantics are not properly defined. For example, COL lacks a model-theoretic semantics since a proper ordering on models is not used. Models are still compared with subset relationship, which has been shown inappropriate in \[9\]. For a given strata in a stratified COL program, there may be many (even infinite) incomparable minimal models based on the simple subset ordering. One of which is selected as the intended semantics. The selection of this particular minimal model is not well justified from the pure model-theoretic point of view. Instead, proof-theoretic point of view is used to justify its selection as the intended semantics. That is, it can be computed bottom-up so it is used as the intended semantics. This approach seems to somewhat depart from the declarative nature of deductive databases.

In LDL, a new ordering, namely d-preferability, is used to replace subset ordering to compare models. Unfortunately, this d-preferability ordering is still not a partial order, based on which the model minimality is ill-defined. In \[8\], a partial order was introduced to compare models. However, such an ordering is too restrictive. A lot of meaningful objects have to be excluded with this ordering. We will examine this issue in Section 4.

Until now, well-defined model-theoretic semantics for nested sets, tuples and relations is still missing in most extensions to Datalog.

In this paper, we propose a novel logic programming based language for nested relational and complex value models called Relationlog which is inspired by LDL. It is a typed extension to Datalog with powerful sets and tuples constructors. Like other Datalog extensions, queries in Relationlog can be evaluated in polynomial space and time. As COL, it directly supports sets and tuples.

The main novel feature of the language is the introduction of powerful mechanisms, namely, partial and complete set terms, for representing and manipulating both partial and complete information on nested sets, tuples and relations. They
generalize of the set grouping and set enumeration mechanisms in LDL and allow
the user to directly encode the open and closed world assumptions on nested sets,
tuples, and relations. They allow directly inference and access to deeply embedded
values in a complex value relation as if the relation is normalized, which greatly in-
creases the ease of use of the language. As a result, the extended relational algebra
operations, as defined in [1, 14, 26] can be represented in Relationlog directly, and
more importantly, recursively in a way similar to Datalog.

Unlike LDL and COL, Relationlog has a well-defined minimal model semantics
which captures the intended semantics for nested sets, tuples and relations. A
stratification in the spirit of a number of other researchers [4, 7, 9] is used. It is
shown that for a stratified Relationlog program, if it is well-typed, then there exists
a minimal model that is preferable to all other models of the program from the
pure model-theoretic point of view. This model can be computed bottom-up using
a finite sequence of fixpoints and therefore, is used as the intended semantics of the
program.

This paper is organized as follows. Section 2 provides an informal introduction
to some of the main features of the language with a number of motivating examples.
Section 3 defines the formal syntax of Relationlog. Section 4 presents its declarative
semantics. Section 5 focuses on the bottom-up semantics. Section 6 summarizes
and points out further research issues.

2. INFORMAL PRESENTATION AND EXAMPLES

In this section, we provide an informal presentation based mainly on examples.
First, we give some motivation for partial and complete set terms and then introduce
the language briefly.

In LDL, set grouping and set enumeration mechanisms are provided to manipu-
late sets. Set grouping is used to construct a set by specifying some property. Set
enumeration is used to construct a set by enumerating its elements.

Example 2.1. Consider the following rules in LDL:

\[
parentof(X, \langle Y \rangle) \leftarrow parentof(X, Y) \\
bookdeal(\{X, Y, Z\}) \leftarrow book(X, Px), book(Y, Py), book(Z, Pz), \\
X \neq Y, X \neq Z, Y \neq Z, Px + Py + Pz < 100
\]

The first rule with set grouping term \( \langle Y \rangle \) groups all parents of a person into a
set in the relation \( parentof \). The second rule with set enumeration term \( \{X, Y, Z\} \)
derives a relation \( bookdeal \) on sets of book titles from the \( book \) relation such that
their total price does not exceed $100.

However, there are three problems with the set grouping mechanism. The first
problem is that it is too primitive and does not allow direct inference and access
to embedded values so that it is cumbersome and ineffective to deal with basic
extended relational algebra operations as shown in the previous section.

The second problem is that the property for grouping is restricted to a single
rule rather than the program (a set of rules).

Example 2.2. If we are given binary relations \( fatherof \) and \( motherof \). We
cannot use the following grouping rules in LDL to group all parents of a person into a set as the grouping we want involves two rules:

\[
\begin{align*}
\text{parentsof}(X, \{Y\}) & \Leftarrow \text{fatherof}(X, Y) \\
\text{parentsof}(X, \{Y\}) & \Leftarrow \text{motherof}(X, Y)
\end{align*}
\]

We have to introduce an intermediate relation such as \text{parentof} first and then use the first rule in Example 2.1 to obtain what we want.

The last problem is that grouping requires stratification which prohibits direct recursive join of two nested relations.

**Example 2.3.** Given binary relation \text{parentsof} with the first argument for a person and second for the set of all parents of the person. We may want to define the \text{ancestorsof} relation recursively in LDL as follows:

\[
\begin{align*}
\text{ancestorsof}(X, \{Y\}) & \Leftarrow \text{parentsof}(X, S), Y \in S \\
\text{ancestorsof}(X, \{Y\}) & \Leftarrow \text{parentsof}(X, S_1), Z \in S_1, \\
& \quad \text{ancestorsof}(Z, S_2), Y \in S_2
\end{align*}
\]

However, the grouping mechanism of LDL requires that the \text{parentsof} and \text{ancestorsof} relations in the body be known before the \text{ancestorsof} relation in the head can be computed. Therefore, direct recursive definition of the relation \text{ancestorsof} is impossible in LDL.

Now we discuss how these problems can be solved. Recall the first grouping rule in Example 2.1. Suppose we are given the following two facts:

\[
\begin{align*}
\text{parentof}(\text{bob}, \text{pam}) \\
\text{parentof}(\text{bob}, \text{tom})
\end{align*}
\]

Then what we can derive with the rule is the fact \text{parentsof}(\text{bob}, \{\text{pam}, \text{tom}\}). If we drop the special meaning of grouping attached to the construct \{Y\} in the rule and simply treat it as a legal term, then we can obtain the following two facts:

\[
\begin{align*}
\text{parentsof}(\text{bob}, \text{pam}) \\
\text{parentsof}(\text{bob}, \text{tom})
\end{align*}
\]

By comparing them with the fact \text{parentsof}(\text{bob}, \{\text{pam}, \text{tom}\}) which we intend to derive, we discover the following:

1. \{\text{pam}\} and \{\text{tom}\} are parts of the set \{\text{pam}, \text{tom}\};
2. \{\text{pam}\} and \{\text{tom}\} together provide sufficient information for us to get the set \{\text{pam}, \text{tom}\};
3. tuples (\text{bob}, \{\text{pam}\}) and (\text{bob}, \{\text{tom}\}) are parts of the tuple (\text{bob}, \{\text{pam}, \text{tom}\});
4. tuples (\text{bob}, \{\text{pam}\}) and (\text{bob}, \{\text{tom}\}) together provide sufficient information for us to get (\text{bob}, \{\text{pam}, \text{tom}\}).

Therefore, the term \{Y\} can be used to provide or denote partial information for a set such that the instantiation of \{Y\} is an element of the set. With this view, a term of the form (X, Y, Z) is then meaningful. The grouping involving several rules such as the ones in Example 2.2 is also meaningful. Furthermore, such a term
can be used not only in the head but also in the body of rules. When used in the head, it derives partial information for a set and complete information for a set can be obtained by combining all such partial results (i.e., grouping). When used in the body, it denotes part of a set. For example, \( \langle Y \rangle \) in the body of a rule is semantically equivalent to \( Y \in S \) for some set \( S \). However, we don’t have to know complete information about this \( S \) when we use \( \langle Y \rangle \) and we don’t even need the predicate \( \in \). Therefore, there is no need to stratify the program based on it. As a result, the recursive join of two nested relations \( \text{parentsof} \) and \( \text{ancestorsof} \) in Example 2.3 can be supported directly with this view as follows:

\[
\begin{align*}
\text{ancestorsof}(X, \langle Y \rangle) & := \text{parentsof}(X, \langle Y \rangle) \\
\text{ancestorsof}(X, \langle Y \rangle) & := \text{parentsof}(X, \langle Z \rangle), \text{ancestorsof}(Z, \langle Y \rangle)
\end{align*}
\]

On the other hand, the set enumeration term \( \{X, Y, Z\} \) of LDL in the second rule of Example 2.1 has a quite different view. It means the set derived with it contains the complete information about the set.

In Relationlog, the construct \( \langle X, \ldots, Z \rangle \) is called a partial set term while \( \{X, \ldots, Z\} \) is called a complete set term in order to reflect their intended meanings as discussed above. They are used in Relationlog program to build-in open and closed world assumptions on sets respectively. Like COL, Relationlog directly supports tuples rather than using functors as LDL. A tuple term of the form \( [X, \ldots, Z] \) is used in Relationlog to represent and manipulate tuples. Since partial and complete set terms as well as tuple terms can embed complex terms, direct access to deeply embedded values is therefore possible in this way. As a result, extended relational algebra operations can be represented directly in Relationlog as simple as in Datalog, see Example 3.3.

Unlike LDL which is untyped, Relationlog is a typed language. There is a notion of schema in Relationlog. Schema in a database corresponds to type declarations in a program. It is important for any database as it provides the description of the database structure and is the basis for storage structure and query optimization strategies. It is essential to the consistency of the database. Its use in Relationlog enables us to get rid of a number of semantic problems as already pointed out in [9].

A Relationlog database consists of two parts: a schema and a program which is a set of rules.

**Example 2.4.** Following is a Relationlog Database.

```
Schema

motherof(String, String)
fatherof(String, String)
parentsof(String, {String})
ancestorsof(String, {String})
```

3. SYNTAX OF RELATIONLOG

We assume the existence of the following pairwise disjoint and possibly countably infinite sets:

1. atomic type symbols $\mathcal{T} = \{\text{Integer, Real, String}\}$;
2. predicate symbols $\mathcal{P}$;
3. constants $\mathcal{C} = I \cup R \cup S$ where $I$ is the set of integers, $R$ is the set of of real numbers, and $S$ is the set of strings;
4. variables $\mathcal{V}$.

**Definition 3.1.** The *types* are defined recursively as follows:

1. $T \in \mathcal{T}$ is an atomic type;
2. if $T_1, ..., T_n$ are types ($n \geq 1$), then $\{T_1, ..., T_n\}$ is a tuple type; and
3. if $T$ is a type, then $\{T\}$ is a set type.

We note $\mathcal{T}^*$ the set of all types.

**Definition 3.2.** Let $p$ be a predicate symbol, and $T_1, ..., T_n$ types. Then $p(T_1, ..., T_n)$ is a relational schema.

As in Datalog, predicate symbols in Relationlog function as relation names.

**Definition 3.3.** A database schema $K$ is a set of relational schemas with distinct predicate symbols.

**Definition 3.4.** The *terms* are defined recursively as follows:

1. a variable $X \in \mathcal{V}$ is either an atomic term, a complete set term, or a tuple term depending on the context.
2. a constant $c \in \mathcal{C}$ is an atomic term;
3. if $O_1, ..., O_n$, ($n \geq 0$) are terms, then $[O_1, ..., O_n]$ is a tuple term;
4. if $O_1, ..., O_n$, ($n > 0$) are distinct terms, then $\langle O_1, ..., O_n \rangle$ is a partial set term;
(5) if $O_1, \ldots, O_n$, ($n \geq 0$) are distinct terms other than partial set terms and are not constructed with partial set terms, then \{\$O_1, \ldots, O_n\} is a complete set term.

We shall adopt the Prolog notation for constants and variables in the examples. That is, a number or string starting with a lower case letter denotes a constant and an identifier starting with an upper case letter denotes a variable.

A term is ground if it has no variables. An individual is a ground atomic term. A tuple is a ground tuple term. A partial set is a ground partial set term. A complete set is a ground complete set term. An object is either an individual, a partial set, a complete set or a tuple.

**Example 3.1.** The following are several examples of objects:

- Individuals: Bob, Pam, 25
- Tuples: [\{Tom, 51\}, [a_1, \{b_1\}], [a_1, \{\{b_1, \{c_1\}\}\}], [a_1, \{\{b_1, \{c_2\}\}, [b_2, \{c_1\}]\}]
- Partial sets: \{Pam, Tom\}, \{\{Bob, 20\}, [Pam, 50]\}
- Complete sets: \{Pam, Tom\}, \{\{Bob, 20\}, [Pam, 50]\}

In the meta language which we use to define Relationlog, we will treat partial sets and complete sets as sets in the traditional sense so that it makes sense to have both $b \in \{a, b, c\}$ and $b \in \{a, b, c\}$.

An object is compact if (1) it is an individual, (2) it is a tuple $[O_1, \ldots, O_n]$ and each $O_i$ is compact for $1 \leq i \leq n$, or (3) it is a complete set \{\$O_1, \ldots, O_n\}. Otherwise, it is non-compact.

In other words, a compact object is not a partial set and doesn’t contain any partial sets.

**Example 3.2.** Given a compact object, let us see how to use partial set terms to denote part of it. For a simple complete set \{\$a, b, c\}, we can use a partial set term $\langle X \rangle$ or $\langle X, Y \rangle$ where $X, Y$ range over the elements in the set. For a set of tuples \{\{a_1, \{b_1, b_2\}\}, [a_2, \{b_1, b_2, b_3\}]\}, we can use a partial set term $\langle X, Y \rangle$ where $X$ ranges over $a_1$ and $a_2$ and $Y$ ranges over the nested sets, or use a partial set term with an embedded partial set term $\langle X, \{Y\}\rangle$ where $X$ still ranges over $a_1$ and $a_2$ but $Y$ ranges over the elements in the two nested sets. Therefore, with embedded partial set terms, we can directly infer and access deeply embedded objects in a complex compact object.

**Definition 3.5.** Let $p \in \mathcal{P}$ be an $n$-ary predicate symbol and $O_1, \ldots, O_n$ are terms. Then $p(O_1, \ldots, O_n)$ is an atom.

An atom $p(O_1, \ldots, O_n)$ is ground if each term $O_i$ is ground for $1 \leq i \leq n$. A ground atom is called a fact. A fact $p(O_1, \ldots, O_n)$ is compact if each object $O_i$ is compact for $1 \leq i \leq n$. Otherwise, it is non-compact.

**Definition 3.6.** A literal is either an atom $p(O_1, \ldots, O_n)$ or a negated atom $\neg p(O_1, \ldots, O_n)$.
**Definition 3.7.** A rule is of the form $A \leftarrow L_1, \ldots, L_n$, where the head $A$ is an atom and the body $L_1, \ldots, L_n$, $n \geq 0$ is a list of literals. A rule is safe if every variable that appears in the head also appears in the body in a non-negated atom.

A fact is just a safe rule with empty body.

**Example 3.3.** Several nested relations defined using extended relational algebra operators in [26] are shown in Figure 1, where $P_1 = \nu_{(B,C,D)}(\nu_{(C,D)}(P_4))$, $P_4 = \mu_{(C,D)}(\mu_{(B,C,D)}(P_1))$, $P_5 = P_1 \cup P_2$, $P_6 = P_1 \cap P_2$, $P_7 = P_1 - P_2$, and $P_8 = P_2 \bowtie P_3$. In Relationlog, we can use following rules to represent these operators.

Nesting: $p_1(X_A, \{X_B, \{X_C, X_D\}\}) \leftarrow p_4(X_A, X_B, X_C, X_D)$

Unnesting: $p_4(X_A, X_B, X_C, X_D) \leftarrow p_1(X_A, \{X_B, \{X_C, X_D\}\})$

Union: $p_6(X_A, \{X_B, \{X_C, X_D\}\}) \leftarrow p_1(X_A, \{X_B, \{X_C, X_D\}\})$

Intersection: $p_8(X_A, \{X_B, \{X_C, X_D\}\}) \leftarrow p_1(X_A, \{X_B, \{X_C, X_D\}\})$

Difference: $p_7(X_A, \{X_B, \{X_C, X_D\}\}) \leftarrow p_1(X_A, \{X_B, \{X_C, X_D\}\}) \land p_2(X_A, \{X_B, \{X_C, X_D\}\})$

Join: $p_9(X_A, X_E, X_B, \{X_C, X_D\}) \leftarrow p_1(X_A, \{X_B, \{X_C, X_D\}\}) \land p_2(X_A, \{X_B, \{X_C, X_D\}\})$

Note that a relation that is obtained by using the nest operations $\nu$ or the unnest operation $\mu$ several times can be unnested or nested respectively in Relationlog with just a single rule. This is possible because the partial set terms in Relationlog can directly access or group deeply nested data.

**Definition 3.8.** A program is a finite set of safe rules.

**Definition 3.9.** A database is a tuple $DB = \langle K, P \rangle$, where $K$ is a database schema and $P$ is a program.

**Definition 3.10.** A query is a list of literals prefixed with `?-`.

We next introduce two syntactic constraints on Relationlog programs, namely well-typed programs and stratified programs.

### 3.1. Well-Typed Programs

Like COL, Relationlog is a typed language. However, COL uses a typed alphabet which is cumbersome from practical point of view. For example, the user has to declare the type for each variable used in the program as in procedural programming languages. Relationlog takes a different approach.

**Definition 3.11.** A type substitution $\Theta$ is a mapping from $C \cup \mathcal{V}$ to $\mathcal{T}$ such that the following hold:

1. if $c \in \mathcal{I}$, then $\Theta(c) = \text{Integer}$
2. if $c \in \mathcal{R}$, then $\Theta(c) = \text{Real}$
3. if $c \in \mathcal{S}$, then $\Theta(c) = \text{String}$
FIGURE 3.1. Nested relations defined with extended relational algebra operators
It is extended to terms, atoms, and literals as follows.

(1) if \( X \in \mathcal{V} \), then \( \Theta X = \Theta(X) \)
(2) if \( c \in \mathcal{C} \), then \( \Theta c = \Theta(c) \)
(3) \( \Theta[O_1, \ldots, O_n] = [\Theta O_1, \ldots, \Theta O_n] \)
(4) \( \Theta\{O_1, \ldots, O_n\} = \{\Theta O_1\} \) if \( \Theta O_1 = \ldots = \Theta O_n \); otherwise, it is undefined
(5) \( \Theta\{O_1, \ldots, O_n\} = \{\Theta O_1\} \) if \( \Theta O_1 = \ldots = \Theta O_n \); otherwise, it is undefined
(6) \( \Theta p(O_1, \ldots, O_n) = p(\Theta O_1, \ldots, \Theta O_n) \)
(7) \( \Theta \neg p(O_1, \ldots, O_n) = p(\Theta O_1, \ldots, \Theta O_n) \)

**Definition 3.12.** Let \( K \) be a schema, \( r \) a safe rule of the form \( A := L_1, \ldots, L_n \).
Then \( r \) is well-typed with respect to \( K \) if and only if there exists a type substitution \( \Theta \) such that \( \Theta L_i \in K \) for \( 1 \leq i \leq n \) implies \( \Theta A \in K \).

**Definition 3.13.** Let \( P \) be a program and \( K \) a schema. Then \( P \) is well-typed with respect to \( K \) if and only if every rule \( r \in P \) is well-typed with respect to \( K \).

**Example 3.4.** Consider the following database:

**Schema**

\[
\begin{align*}
p(\text{Integer}, \{\text{String}\}) \\
q(\text{String}, \text{Integer})
\end{align*}
\]

**Program**

\[
\begin{align*}
p(Y, \{X\}) := q(X, Y) \\
q(\text{john}, \text{mary})
\end{align*}
\]

The rule is well-typed with respect to the schema but the fact is not. Therefore the program is not well-typed.

In LDL, rules like \( p(\{X\}) := p(X) \) cause problems and are excluded by using stratification. In RelationLog, they are also excluded, but by the typing constraint instead.

**3.2. Stratified Programs**

The notion of stratification has been used in order to give semantics to programs involving negation and sets \([4, 7, 9]\). We present a similar notion here.

**Definition 3.14.** Let \( DB = \langle K, P \rangle \) be a database. We note \( D_P \) the set of all predicate symbols appearing in \( K \). The relationships \( >, \) and \( \geq \) on \( D_P \) are defined as follows:

(1) \( p > q \) if there is a rule in which \( p \) is in the head, \( q \) in the body, and \( q \) either is in a negative literal, or contains complete set terms.
(2) \( p \geq q \) if there is a rule in which \( p \) is in the head, \( q \) is in the body, and \( p > q \) is not true.

They are extended by the following transitivity rules:
(1) \( p > q \) if there exists \( r \) such that \( p \geq r \) and \( r > q \).
(2) \( p \geq q \) if there exists \( r \) such that \( p \geq r \) and \( r \geq q \).

Let \( p(O_1, ..., O_n) \) be in the body of a rule, and assume that \( O_k \) is or contain a complete set term for some \( k \in \{1, ..., n\} \). Then when we use the rule to infer information, the value for \( O_k \) must be completely determined. Therefore, we require the predicate symbol in the head to be at a higher level. The case for negation is the same.

**Definition 3.15.** For each program \( P \), the dependency graph \( G_P \) is a marked graph constructed as follows:

1. the set of nodes is \( D_P \),
2. there is an edge from \( p \) to \( q \) if \( p \geq q \), and
3. there is a marked edge from \( p \) to \( q \) if \( p > q \).

A dependency graph of \( P \) represents the dependency relationship between predicate symbols of \( P \).

**Definition 3.16.** A program \( P \) is stratified if and only if its dependency graph \( G_P \) has no cycle with a marked edge.

An alternative definition may be obtained by using the relationship > as follows.

**Proposition 3.1.** A program \( P \) is stratified if and only if the relationship > is both transitive and irreflexive.

Since a given program has only a finite number of predicate symbols, it can be statically determined whether a program is stratified or not.

**Example 3.5.** Consider the program in the following Relationlog database.

<table>
<thead>
<tr>
<th>Schema</th>
<th>( p({\text{String}}) )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( q(\text{String}) )</td>
</tr>
<tr>
<td></td>
<td>( r(\text{String}, {\text{String}}) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Program</th>
<th>( p({X}) \rhd q(X) )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( q(X) \rhd p(S), r(X, S) )</td>
</tr>
<tr>
<td></td>
<td>( q(a) )</td>
</tr>
<tr>
<td></td>
<td>( q(b) )</td>
</tr>
<tr>
<td></td>
<td>( r(c, {a, b}) )</td>
</tr>
</tbody>
</table>

This program is not stratified as we have a marked edge in the cycle \( p \rightarrow q \rightarrow p \).

**Example 3.6.** The program in the following Relationlog database is stratified.

<table>
<thead>
<tr>
<th>Schema</th>
<th>( p({\text{String}}) )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( q({\text{String}}) )</td>
</tr>
<tr>
<td></td>
<td>( r(\text{String}) )</td>
</tr>
</tbody>
</table>
Program  
\[ p((X)) \rightarrow q(X) \]
\[ q((X)) \rightarrow r(X) \]
\[ q(\{a\}) \]
\[ r(a) \]
\[ r(b) \]

The stratification of the program induces an order of evaluation of the predicate symbols as follows.

**Proposition 3.2.** Let \( P \) be a program, and \( D_P \) be the set of all predicate symbols of \( P \). Then \( P \) is stratified if and only if there is a partition \( D_P = D_1 \cup \ldots \cup D_n \) such that

1. if \( p \in D_i \) and \( p > q \), then there exists a \( j \) such that \( i > j \) and \( q \in D_j \),
2. if \( p \in D_i \) and \( p \geq q \), then there exists a \( j \) such that \( i \geq j \) and \( q \in D_j \).

**Proof.** Straightforward from Definition 16. \( \square \)

For example, the program in Example 3.6 can have a partition: \( D_P = \{p, q\} \cup \{r\} \) while the program in Example 3.5 cannot have such a partition.

Each partition of the predicate symbols induces a partition of the program into strata. For each \( D_P = D_1 \cup \ldots \cup D_n \), let \( P = P_1 \cup \ldots \cup P_n \), where for each \( i \),

\[ P_i = \{ r \in P \mid \text{the predicate symbol in the head of } r \text{ is in } D_i \} \]

Continue with the program in Example 3.6, we have \( P = P_1 \cup P_2 \) where

\[ P_1 = \{ q((X)) \rightarrow r(X), q(\{a\}), r(a), r(b) \} \]
\[ P_2 = \{ p((X)) \rightarrow q(X) \} \]

4. **DECLARATIVE SEMANTICS OF RELATIONLOG**

In this section, we define the Herbrand interpretations and models for RelationLog programs. Since we allow tuple terms and partial and complete set terms in our programs, we need to define the universe so that tuples, partial sets and complete sets are elements of the universe.

For each atomic type \( T \), we associate a set of constants with it. This set is called the **domain** of \( T \), and denoted by \( \text{dom}(T) \). In particular, we have \( \text{dom}($Integer$) = \mathcal{I}, \text{dom}($Real$) = \mathcal{R}, \text{dom}($String$) = \mathcal{S} \).

The domains of constructed types are obtained as follows:

1. for a tuple type \([T_1, \ldots, T_n] \), \( \text{dom}([T_1, \ldots, T_n]) = \text{dom}(T_1) \times \ldots \times \text{dom}(T_n) \);
2. for a set type \( \{T\}, \text{dom}(\{T\}) = \mathcal{P}(\text{dom}(T)) \).

where

\[ \mathcal{P}(S) = \{ \{o_1, \ldots, o_n\} \mid n > 0 \text{ and } o_i \in S \text{ for } i = 1, \ldots, n \} \cup \{ \{o_1, \ldots, o_n\} \mid n \geq 0 \text{ and } o_i \in S \text{ is compact for } i = 1, \ldots, n \} \]

**Definition 4.1.** The **universe** \( U \) of objects is defined as \( U = \cup_{T_i \in \mathcal{T}} \text{dom}(T_i) \)

**Definition 4.2.** Let \( K \) be a schema. The **Herbrand base** \( B_K \) for \( K \) is the set of all facts which can be formed by using the predicate symbols appearing in \( K \).
and objects in the universe $U$.

**Definition 4.3.** Let $K$ be a schema. A fact $p(o_1, \ldots, o_n) \in B_K$ is well-typed with respect to $K$ if there exists a relational schema $p(T_1, \ldots, T_n) \in K$, such that $o_i \in \text{dom}(T_i)$ for $1 \leq i \leq n$. A set $S \subseteq B_P$ is well-typed with respect to $K$ if and only if every atom $A \in S$ is well-typed with respect to $K$. It is compact if and only if every atom $A \in S$ is compact.

**Definition 4.4.** Let $DB = \langle K, P \rangle$ be a database. An interpretation of $P$ is a compact and well-typed subset of $B_K$ with respect to $K$.

Note that although partial information about sets can be inferred or used in a RelationLog program, an interpretation does not contain any partial information. It is complete and provide precise interpretation of various constructs in the program.

### 4.1 Satisfaction

In order to define the notion of satisfaction, we introduce the following auxiliary notions.

**Definition 4.5.** An object $o$ is part-of a compact object $d'$, denoted by $o \lessdot d'$, if and only if

1. $o = o'$;
2. $o$ is a partial set and for each $o_i \in o$ there exists $d'_i \in d'$ such that $o_i \lessdot d'_i$; or
3. $o = [o_1, \ldots, o_n]$ and $o' = [d'_1, \ldots, d'_n]$ such that $o_i \lessdot d'_i$ for $1 \leq i \leq n$.

**Definition 4.6.** A fact $p(o_1, \ldots, o_n)$ is part-of a compact fact $p(d'_1, \ldots, d'_n)$, denoted by $p(o_1, \ldots, o_n) \lessdot p(d'_1, \ldots, d'_n)$, if and only if $o_i \lessdot d'_i$ for $1 \leq i \leq n$.

The following are several examples:

$a \lessdot a$

$\langle a_1, a_2 \rangle \lessdot \{a_1, a_2, a_3\}$

$\{a_1, a_2, a_3\} \lessdot \{a_1, a_2, a_3\}$

$[a_1, \{b_1, b_2\}] \lessdot \{a_1, \{b_1, b_2, b_3\}\}$

$[a_1, \{b_1, b_2, b_3\}] \lessdot [a_1, \{b_1, b_2, b_3\}]$

$p(a_1, \{[b_1, \{c_1\}]\}) \lessdot p(a_1, \{[b_1, \{c_1\}], [b_2, \{c_1, c_2\}]\})$

$p(a_1, \{[b_1, \{c_1\}]\}) \lessdot p(a_1, \{[b_1, \{c_1\}], [b_2, \{c_1, c_2\}]\})$

$p(a_1, \{[b_2, \{c_1\}]\}) \lessdot p(a_1, \{[b_1, \{c_1\}], [b_2, \{c_1, c_2\}]\})$

**Definition 4.7.** A set $S$ of facts is part-of a compact fact $A'$, denoted by $S \lessdot A'$, if and only if for each $A \in S$, $A \lessdot A'$.

**Definition 4.8.** A ground substitution $\theta$ is a mapping from $V$ to $U$. It is extended to terms, atoms, and literals as follows:

1. if $X \in V$ then $\theta X = \theta(X)$
2. if $c \in C$ then $\theta c = c$
(3) $\theta[O_1, \ldots, O_n] = [\theta O_1, \ldots, \theta O_n]$
(4) $\theta(O_1, \ldots, O_n) = (\theta O_1, \ldots, \theta O_n)$
(5) $\theta\{O_1, \ldots, O_n\} = \{\theta O_1, \ldots, \theta O_n\}$
(6) $\theta p(O_1, \ldots, O_n) = \theta(O_1, \ldots, O_n)$.
(7) $\theta A = \neg \theta A$.

**Definition 4.9.** Two objects $o$ and $o'$ from the same domain are compatible if and only if one of the following holds:

1. both are constants and are equal;
2. $o = [o_1, \ldots, o_n]$ and $o' = [o'_1, \ldots, o'_n]$ such that $o_i$ and $o'_i$ are compatible for $1 \leq i \leq n$;
3. both are partial sets;
4. $o$ is a partial set and $o'$ is a complete set such that $o \prec o'$;
5. both are complete sets and are equal.

**Definition 4.10.** Two facts $p(o_1, \ldots, o_n)$ and $p(o'_1, \ldots, o'_n)$ are compatible if and only if $o_i$ and $o'_i$ are compatible for $1 \leq j \leq n$.

The following pairs of objects and facts are compatible:

- $a$ and $a$
- $\langle a_1 \rangle$ and $\langle a_1, a_3 \rangle$
- $\langle a_1 \rangle$ and $\{a_1, a_2, a_3\}$
- $[a, \langle b_1, b_2 \rangle]$ and $[a, \{b_1\}]$
- $[\langle a_1 \rangle, \{b_1, b_2\}]$ and $[\langle a_2 \rangle, \{b_1\}]$
- $p(\langle a_1 \rangle)$ and $p(\langle a_1, a_2 \rangle)$
- $p(\langle a_1 \rangle, \{b_1\})$ and $p(\{a_1, a_2\}, \{b_1\})$

However, the following pairs are not compatible:

- $a_1$ and $a_2$
- $\langle a_1 \rangle$ and $\{a_2, a_3\}$
- $p(a, \langle b_1 \rangle)$ and $p(a, \{b_1\})$

A set of objects or facts are compatible if and only if each pair of them are compatible.

**Definition 4.11.** Let $P$ be a program and $I$ an interpretation of $P$. The notion of satisfaction (denoted by $\models$) and its negation (denoted by $\not\models$) are defined as follows.

1. For each fact $A$, $I \models A$ if and only if there exists $A' \in I$ such that $A \prec A'$.
2. For each ground negative literal $\neg A$, $I \models A$ if and only if $I \not\models A$.
3. Let $r$ be a rule of the form $A \rightarrow L_1, \ldots, L_n$. Then $I \models r$ if and only if for every ground substitution $\theta$, $I \models \theta L_1, \ldots, I \models \theta L_n$, implies $I \models \theta A$.
4. For the program $P$, $I \models P$ if and only if the following hold:
   (a) for every rule $r \in P$, $I \models r$
   (b) for all instantiated rules satisfied by $I$ if their heads are compatible, then there exists a fact in $I$ such that each head is part-of.
Definition 4.12. Let \( P \) be a program. A model \( M \) of \( P \) is an interpretation which satisfies \( P \).

Example 4.1. Consider the following mono-stratum program:

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

The following interpretation \( I \) is a model of the program

\[
I = \{r(a), q(b), p(\{a\}), p(\{a, b\})\}
\]

It does capture our intention to treat partial set term \( \langle X \rangle \) in the head of the two rules as a generalized grouping mechanism.

Should we drop the condition (b) in (4) of Definition 11, then the following interpretation \( I' \) which does not treat partial set term \( \langle X \rangle \) as grouping at all would be a model.

\[
I' = \{r(a), q(b), p(\{a\}), p(\{b\})\}
\]

Furthermore, \( I' \) is even smaller than \( I \) based on the ordering we will introduce shortly.

Unlike LDL in which some programs may not have a model, for a well-typed Relationlog program \( P \) with respect to a schema \( K \), \( B_k \) is a trivial model of \( P \).

Proposition 4.1. A well-typed program has a model.

4.2. Comparing Models

Like Datalog, a Relationlog program may have many different models.

Example 4.2. Consider the following mono-stratum program:

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

Possible models for this program are

\[
\begin{align*}
M_1 = \{q(a), p(\{a\})\} \\
M_2 = \{q(a), q(b), p(\{a, b\})\} \\
M_3 = \{q(a), q(c), p(\{a, c\})\} \\
M_4 = \{q(a), q(b), q(c), p(\{a, b, c\})\}
\end{align*}
\]

They all interpret grouping properly. Their intersection is \( \{q(a)\} \) which is not a model.

As shown in the above example, the simple intersection of models is not nec-
essarily a model because of nested sets and tuples. Therefore, we cannot use the traditional model intersection to obtain the minimal model for a program. As discussed in [9], generalized intersection won’t work either. Therefore, we use an ordering on objects and models based on the ordering proposed in [19] so that we can properly define the notion of minimal model from pure model-theoretic point of view.

**Definition 4.13.** Let \( o \in U \) and \( o' \in U \) be two compact objects. Then \( o \) is a **sub-object** of \( o' \), denoted by \( o \preceq o' \), if and only if one of the following holds:

1. both are constants and are equal;
2. \( o = [o_1, \ldots, o_n] \), \( o' = [o'_1, \ldots, o'_n] \), and for \( 1 \leq i \leq n \), \( o_i \preceq o'_i \);
3. both are complete sets and for each \( o_i \in o - o' \), there exists \( o'_i \in o' - o \), such that \( o_i \preceq o'_i \).

Note that the sub-object relationship is only defined between compact objects. The reason is that we just use it to compare models which contain only compact objects.

**Example 4.3.** Following are several examples:

\[
\begin{align*}
\{\{a\}, \{a, b\}\} & \preceq \{\{a\}, \{b\}, \{a, b\}\}, \\
\{\{a\}, \{a, b\}\} & \preceq \{\{a\}, \{b\}, \{a, b\}\} \\
\{\{a\}, \{a\}, \{a, b\}\} & \not\succeq \{\{a\}, \{a, b\}\}, \\
\{\{a\}, \{a\}, \{a, b\}\} & \not\succeq \{\{a\}, \{a, b\}\}.
\end{align*}
\]

We have the following important result.

**Proposition 4.2.** The sub-object relationship is a partial order.

**Proof.** Straightforward from the proof in [19] \( \square \)

In [8], the sub-object relationship is defined in the way similar to ours except that item (3) is replaced with the following:

3. both are sets and for each \( o_i \in o \), there exists \( o'_i \in o' \), such that \( o_i \preceq o'_i \).

As a result, such a sub-object relationship is only a pre-order. In order to make it an order, the notion of **reduced objects** is introduced. Therefore, meaningful objects like \( \{\{a\}, \{a, b\}\} \) and \( \{\{a\}, \{a, b\}\} \) are not reduced and could not exist in the program. In HILOG [11], a similar notion was used which suffers from the same problem. Our notion of sub-object doesn’t have this problem.

**Definition 4.14.** A compact fact \( p(o_1, \ldots, o_n) \in U \) is a **sub-fact** of a compact fact \( p(o'_1, \ldots, o'_n) \in U \), denoted by \( p(o_1, \ldots, o_n) \preceq p(o'_1, \ldots, o'_n) \), if and only if \( o_i \preceq o'_i \) for \( 1 \leq i \leq n \).

**Proposition 4.3.** The sub-fact relationship is a partial order.

**Proof.** Straightforward from Proposition 4.2. \( \square \)

**Proposition 4.4.** Let \( A, B \) be two compact facts such that \( A \preceq B \). Then the
following hold:

1. If $A$ and $B$ contains no sets, then $A = B$.
2. If $C \triangleleft A$, then $C \triangleleft B$.

**Proof.** Straightforward from Definitions 14 and 6.

**Definition 4.15.** Let $M_1$ and $M_2$ be two models of $P$. Then $M_1$ is a sub-model of $M_2$, denoted by $M_1 \preceq M_2$, if and only if for every fact $A \in M_1 - M_2$, there exists a fact $A' \in M_2 - M_1$ such that $A \preceq A'$.

**Proposition 4.5.** Let $L_P$ be the set of all models of the program $P$. Then $\preceq$ is a partial order on $L_P$.

**Proof.** Straightforward from Proposition 4.3 and Definition 15.

**Definition 4.16.** A model $M$ of $P$ is minimal if and only if for each model $N$ of $P$, if $N \preceq M$ then $N = M$.

For the models $M_1$, $M_2$, $M_3$ and $M_4$ in Example 4.2, $M_1$ is minimal.

The notion of sub-model is somewhat similar to the notion of d-preferability ($\preceq_d$) in LDL [9]. However, the model minimality in LDL is ill-defined based on $\preceq_d$. First, $\preceq_d$ is irreflexive by definition. Besides, $\preceq_d$ is not transitive since nested infinite sets are allowed. Finally, $\preceq_d$ is not anti-symmetry since there isn’t an equivalent notion of sub-object. For example, let $M_1 = \{p(\{1\}, \{1, 2\})\}$ and $M_2 = \{p(\{1, 2\})\}$. Then $M_1 \preceq_d M_2 \preceq_d M_1$, but $M_1 \neq M_2$.

### 4.3. Perfect Model Semantics

A multi-strata Relationlog program may have more than one minimal model.

**Example 4.4.** Consider the following two-strata program:

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

The following are two minimal models which are not comparable under $\preceq$:

- $M_1 = \{r(a), q(\{a\}), p(a)\}$
- $M_2 = \{r(a), q(\{a, b\})\}$

Of all minimal models of a program, we intend to choose a well-justified minimal model, from pure model theoretic point of view, as the intended semantics of the program. Thus, we extend the notion of perfect model originally proposed in [22] as follows.

**Definition 4.17.** Let $P$ be a stratified program, $M$ and $N$ be distinct models of $P$. Then $M$ is preferable to $N$, denoted by $M \ll N$, if for every fact $p(o_1, ..., o_n) \in$
$M - N$ there is a fact $q(o_1, \ldots, o_n) \in N - M$, such that $p > q$ or $p(o_1, \ldots, o_n) \preceq q(o_1, \ldots, o_n)$.

In other words, $M$ compared to $N$, minimizes the extension of predicates that appear in lower strata at the expense of adding facts of predicates that appear in higher strata.

Continue with the Example 4.4, $M_1$ is preferable to $M_2$.

**Proposition 4.6.** Preferability is a transitive relation on models.

**Proof.** Straightforward generation of the proof in [22]. □

**Definition 4.18.** A model $P$ is **perfect** if it is preferable to all models of $P$.

Therefore, the perfect model in Relationlog captures not only the notion of prioritized minimization, but also the semantics of nested sets and tuples, which is an open problem left in the semantics of LDL.

For the program in Example 4.4, $M_1$ is the perfect model.

Consider another two-strata program:

$$p((X)) := q(X), \neg r(X)$$
$$q(a)$$
$$q(b)$$
$$q(c)$$
$$r(a)$$

The following are three minimal models which are not comparable under $\preceq$:

$$M_1 = \{q(a), q(b), q(c), r(a), p(\{b, c\})\}$$
$$M_2 = \{q(a), q(b), q(c), r(a), r(b), p(\{c\})\}$$
$$M_3 = \{q(a), q(b), q(c), r(a), r(b), r(c)\}$$

We have $M_1 \ll M_2 \ll M_3$ and $M_1$ is the perfect model.

The definition 18 does not imply that the perfect model of a program exists. The issue of existence of the perfect model will be settled in the next section.

**Definition 4.19.** Let $P$ be a program. Then its **declarative semantics** is given by the perfect model if it exists.

**Definition 4.20.** Let $P$ be a program and $Q$ a query of the form $\exists L_1, \ldots, L_n$. Assume that the perfect model $M$ exists. Then an **answer** to $Q$ based on $P$ is a ground substitution $\theta$ such that $M \models \theta L_i$ for $1 \leq i \leq n$.

5. **BOTTOM-UP SEMANTICS**

In this section, we show that for a well-typed stratified Relationlog program, the unique perfect model exists and can be constructed bottom-up.
Definition 5.1. Let $P$ be a well-typed program with respect to $K$ and $I$ an interpretation of $P$. The operator $T_P$ over $I$ is defined as follows.

$$T_P(I) = \{ \theta A \mid A :: L_1, \ldots, L_n \in R, \text{ and there exists a ground substitution } \theta \text{ such that } I \models \theta L_1, \ldots, I \models \theta L_n \}$$

Note that $T_P$ is similar to the traditional immediate consequence operator in logic programming. Unlike Datalog and LDL, the result of $T_P$ is not necessarily an interpretation.

Example 5.1. Let $I = \{ p(b, \{b,c\}) \}$. Consider the following database.

Schema $p(String, \{String\})$

Program $p(a, \langle X \rangle) :: p(b, \langle X \rangle)$
$p(a, \langle a \rangle)$

Then $T_P(I) = \{ p(a, \langle a \rangle), p(a, \langle b \rangle), p(a, \langle c \rangle) \}$, which is not an interpretation as it contains partial sets.

5.1. Grouping

As discussed in Section 2, we deliberately allow atoms with partial sets to be inferred in Relationlog as intermediate results. They must be grouped properly. Therefore, in what follows we discuss how to group atoms. First, we introduce the following auxiliary notion.

Definition 5.2. Let $S$ be a set of facts and $S'$ a compatible subset of $S$. Then $S'$ is a maximal compatible set in $S$ if there does not exist a fact in $S - S'$ that is compatible with each object in $S'$.

Example 5.2. Let $S = \{ p(\langle a_1 \rangle), p(\langle a_2 \rangle), p(\langle a_3 \rangle), p(\{a_1, a_2, a_3\}) \}$ be a set of facts. Then $S_1 = \{ p(\langle a_1 \rangle), p(\langle a_2 \rangle), p(\langle a_3 \rangle) \}$ and $S_2 = \{ p(\langle a_1 \rangle), p(\langle a_2 \rangle), p(\{a_1, a_2\}) \}$ are two maximal compatible sets in $S$.

Definition 5.3. The grouping operator $G$ is defined recursively on a set of compatible objects as follows:

1. If $o \in S$ is compact, then $G(S) = o$.
2. If $S$ is a set of partial sets, then $G(S) = \{ G(S') \mid S' = \{ o \mid o \in s, \text{ and } s \in S \} \}$ is a maximal compatible set of objects.
3. If $S$ is a set of tuples of the form $[o_1, \ldots, o_n]$ without a compact element, then $G(S) = [G(S_1), \ldots, G(S_n)]$ where $S_i = \{ o_i \mid [o_i, \ldots, o_n] \in S \}$ for $1 \leq i \leq n$.

It is extended to a set of facts as follows:
Example 5.3. Following examples show how the grouping operator can be applied to objects. The last two examples are complex and are therefore shown in an intuitive way.

\[ G(\{a\}) = a \quad G(\{a\}) = [a] \quad G(\{a\}) = \{a\} \]

\[ G(\{a_1, a_2, a_3\}) = \{a_1, a_2, a_3\} \]

\[ G(\{a_1, a_2, a_3\}) = G(\{a_1\}), G(\{a_2\}) = \{a_1, a_3\} \]

\[ G(\{a, b_1, b_2\}) = G(\{a\}), G(\{b_1, b_2\}) = [a_1, \{b_1, b_2\}] \]

\[ G(\{a_1, b_1, b_2\}) = G(\{a_1\}), G(\{b_1, b_2\}) = [a_1, \{b_1, b_2\}] \]

\[ G(\{a_1, b_1\}) = G(\{a_1\}), G(\{b_1\}) = [a_1, \{b_1\}] \]

\[ G(\{a_1, b_1\}) = G(\{a_1\}), G(\{b_1\}) = [a_1, \{b_1\}] \]

\[ G(\{a_1, b_1\}) = G(\{a_1\}), G(\{b_1\}) = [a_1, \{b_1\}] \]

\[ G(\{a_1, b_1\}) = G(\{a_1\}), G(\{b_1\}) = [a_1, \{b_1\}] \]

\[ G(\{a_1, b_1\}) = G(\{a_1\}), G(\{b_1\}) = [a_1, \{b_1\}] \]

\[ G(\{a_1, b_1\}) = G(\{a_1\}), G(\{b_1\}) = [a_1, \{b_1\}] \]

Example 5.4. Following examples show how the grouping operator can be applied to facts:

\[ G(p(a_1)), = p(G(\{a_1\}, \{a_1, a_2\})) = p(\{a_1, a_2\}) \]

\[ G(p(a_1)), = p(G(\{a_1\}, \{a_1, a_2\})) = p(\{a_1, a_2\}) \]

\[ G(p(a_1), b_1) = p(G(\{a_1\}, \{b_1\}, \{b_2\})) = p(a_1, \{b_1, b_2\}) \]

\[ G(p(a_1), b_1) = p(G(\{a_1\}, \{b_1\}, \{b_2\})) = p(a_1, \{b_1, b_2\}) \]

\[ G(p(a_1), b_1) = p(G(\{a_1\}, \{b_1\}, \{b_2\})) = p(a_1, \{b_1, b_2\}) \]

\[ G(p(a_1), b_1) = p(G(\{a_1\}, \{b_1\}, \{b_2\})) = p(a_1, \{b_1, b_2\}) \]

\[ G(p(a, b_1)) = p(G(\{a, b_1\})) = p(a, \{b_1\}) \]

\[ p(a, \{b_1\}) \cup p(G(\{a, b_2\})) = p(a, \{b_2\}) \]
\[ G\{p(a_1, b_1)\} = \{G\{p(a_1, b_1)\}, = \{p(a_1, \{b_1, b_2\}\} = \{p(a_1, \{b_1, b_2\}\)}, p(a_1, \{b_2\}), p(a_1, \{b_2\}\}) = q(\{a_1, a_2, a_3\}), \} q(\{a_1, a_2, a_3\}), \} r(\{b_1, b_2\}), \} r(\{b_1, b_2\}) \} \]

As shown in Example 5.1, partial set terms in Relationlog can function in two different ways depending on whether they are in the head of rules or in the body of rules. When in the head, they are used to accumulate partial information for the corresponding complete sets. The conversion from partial sets to complete sets is done with the grouping operator \( G \) as shown in Examples 5.3 and 5.4. When in the body, they are used to denote part of the corresponding complete sets. The conversion from complete sets to the corresponding partial sets is captured by the notion part-of \((\leq)\).

The main purpose of introducing the grouping operator \( G \) is to convert \( TP(I) \) into an interpretation. Continue with the Example 5.1, \( G(TP (I)) = \{p(a, \{a, b, c\})\} \) which is an interpretation of the program.

**Proposition 5.1.** The grouping operator \( G \) has the following properties.

1. Let \( I \) be a set of facts. If \( A \in G(I) \), then there is a maximal compatible subset \( S \) of \( I \) such that \( S \subseteq A \) and \( G(S) = \{A\} \).
2. Let \( S \) and \( S' \) be two compatible sets of facts such that \( G(S) = \{A\}, G(S') = \{A'\} \). Then \( S \subseteq S' \) implies \( A \leq A' \).
3. Let \( S \) be a compatible set of facts such that \( G(S) = \{A\} \) and \( A' \) a compact fact. If \( S \subseteq A' \), then \( A \leq A' \).
4. If \( I \) is an interpretation, then \( G(I) = I \).
5. Let \( I \) and \( J \) be sets of facts. If the predicate symbols in \( I \) and \( J \) are disjoint, then \( G(I \cup J) = G(I) \cup G(J) \).
6. Let \( I \) and \( J \) be sets of facts. If \( I \subseteq J \) then \( G(I) \leq G(J) \).

**Proof.** (1) - (5) Straightforward from Definitions 6, 7, and 3. (6) Let \( A \in G(I) \) - \( G(J) \). Then \( A \) is compact. If \( A \in I \), then \( A \in J \) since \( I \subseteq J \). Thus, \( A \in G(J) \), which is a contradiction. Therefore, \( A \notin I \). By (1), there exists a maximal compatible subset \( S \) of \( I \) such that \( S \subseteq A \) and \( G(S) = \{A\} \). Since \( I \subseteq J \), there exists a maximal compatible subset \( S' \) of \( J \) such that \( S \subseteq S' \) and \( G(S') = \{A'\} \subseteq G(J) \). By (2), \( A \leq A' \). As \( A \notin G(J) \), \( A \notin A' \) and \( S \notin S' \). Since \( S \) is a maximal compatible set of \( I \) and \( S \subseteq S' \), \( S' \notin I \) and \( A' \notin G(J) \). Therefore, \( A' \in G(J) \) - \( G(I) \). We have \( G(I) \leq G(J) \). \( \square \)

The operators \( G \) and \( TP \) together function as the traditional immediate consequence operator in logic programming. They have the following property.

**Proposition 5.2.** Let \( DB = \langle K, P \rangle \) be a database and \( M \) a well-typed interpretation of \( P \) with respect to \( K \). If \( M \) is a model of \( P \), then \( G(TP(M)) \leq M \).

**Proof.** Let \( B \in G(TP(M)) - M \). Then \( B \) is compact. We first prove \( B \notin TP(M) \). Suppose \( B \in TP(M) \), then there exists a rule \( A \vdash L_1, \ldots, L_n \) and a ground substitution \( \theta \) such that \( M \models \theta L_1, \ldots, M \models \theta L_n \) and \( B = \theta A \). Since \( M \) is a model of \( P \), \( M \models \theta L_1, \ldots, L_n \) and \( B = \theta A \). Since \( M \) is a model of \( P \), \( M \models B \). As \( B \) is compact, \( B \in M \) by Definitions 11, 6 and 5, which a
contradiction.

Since \( B \not\models T_P(M) \), there exists a maximal compatible subset \( S \) of \( T_P(M) \) such that \( S \prec B \) and \( G(S) = \{ B \} \) by Proposition 5.1 (1). For each \( B' \in S \), there exists a rule \( A : L_1, \ldots, L_n \) and a ground substitution \( \theta \) such that \( M \models \theta L_1, \ldots, M \models \theta L_n \) and \( B' = \theta A \). Since \( M \) is a model of \( P \), \( M \models B' \). As \( S \) is a compatible set, there exists a \( B'' \in M \) such that for each \( B' \in S \), \( B' \prec B'' \) by Definition 11 (4). That is, \( S \prec B'' \). By Proposition 5.1 (3), \( B \preceq B'' \). As \( B \not\models M \), \( B \not\models B'' \). Since \( S \) is a maximal compatible set of \( T_P(M) \) and \( B \subset B'' \), \( B'' \not\models G(T_P(M)) \). Therefore \( B'' \in M - G(T_P(M)) \). We have \( G(T_P(M)) \preceq M \). \( \square \)

**Example 5.5.** Consider the following database:

**Schema**

\[ p(\{ 'String' \}) \]
\[ q( 'String' ) \]
\[ r( 'String' ) \]

**Program**

\[ p(\{ X \}) : = q( X ) \]
\[ p(\{ X \}) : = r( X ) \]
\[ p( a ) \]
\[ q( b ) \]

Let \( M = \{ r( a ), q( b ), p(\{ a, b, c \}) \} \). Then \( M \) is a model and the following holds:

\[ G(T_P(M)) = \{ r( a ), q( b ), p(\{ a, b \}) \} \preceq \{ r( a ), q( b ), p(\{ a, b, c \}) \} = M. \]

However, the converse is not true. That is, if \( G(T_P(I)) \preceq I \), \( I \) is not necessarily a model.

**Example 5.6.** Consider the following database:

**Schema**

\[ p(\{ 'String' \}) \]

**Program**

\[ p(\{ a, b \}) \]

Let \( I = \{ p(\{ a, b, c \}) \} \). Then \( I \) is a well-typed interpretation. We have \( G(T_P(I)) = \{ p(\{ a, b \}) \} \preceq I \). However, \( I \) is not a model of the program.

In Relationlog, \( G \) and \( T_P \) together function as the traditional immediate consequence operator in logic programming.

### 5.2. Bottom-Up Semantics

We proceed to show that for a well-typed stratified program, the perfect model can be constructed bottom-up using a sequence of fixpoint operators.

**Definition 5.4.** An interpretation \( I \) of \( P \) is supported if and only if \( I \preceq G(T_P(I)) \).

**Proposition 5.3.** Let \( DB = \langle K, P \rangle \) be a database and \( M \) a model of \( P \). Then \( M \) is a supported model of \( P \) if and only if \( M = G(T_P(M)) \).
Proof. Straightforward from Definition 5.3 and Proposition 5.2. □

In a supported model, every fact can be inferred.

Definition 5.5. The powers of the operator $T_P$ are defined using the grouping operator as follows:

$$T_P \uparrow 0(I) = I$$
$$T_P \uparrow n(I) = T_P(G(T_P \uparrow (n - 1)(I))) \cup T_P \uparrow (n - 1)$$
$$T_P \uparrow \omega(I) = \bigcup_{n=0}^{\infty} T_P \uparrow n(I)$$

Example 5.7. Consider the following Relationlog database with a mono-stratum program.

Schema

$reach(String, \{String\})$
$edge(String, \{String\})$

Program

$reach(X, \{Y\}) \leftarrow edge(X, \{Z\})$
$reach(X, \{Y\}) \leftarrow edge(X, \{Z\}), reach(Z, \{Y\})$
$edge(a, \{b\})$
$edge(b, \{c\})$
$edge(c, \{d, e\})$

Then we have

$T_P \uparrow 0(\{\}) = \{\}$
$T_P \uparrow 1(\{\}) = \{edge(a, \{b\}), edge(b, \{c\}), edge(c, \{d, e\})\}$
$T_P \uparrow 2(\{\}) = \{edge(a, \{b\}), edge(b, \{c\}), edge(c, \{d, e\}),$
$reach(a, \{b\}), reach(b, \{c\}), reach(c, \{d\}), reach(c, \{e\})\}$
$T_P \uparrow 3(\{\}) = \{edge(a, \{b\}), edge(b, \{c\}), edge(c, \{d, e\}),$
$reach(a, \{b\}), reach(b, \{c\}), reach(c, \{d\}), reach(c, \{e\}),$
$reach(a, \{c\}), reach(b, \{d\}), reach(b, \{e\})\}$
$T_P \uparrow 4(\{\}) = \{edge(a, \{b\}), edge(b, \{c\}), edge(c, \{d, e\}),$
$reach(a, \{b\}), reach(b, \{c\}), reach(c, \{d\}), reach(c, \{e\}),$
$reach(a, \{c\}), reach(b, \{d\}), reach(b, \{e\}),$
$reach(a, \{d\}), reach(a, \{e\})\}$
$T_P \uparrow \omega(\{\}) = T_P \uparrow 4(\{\})$

Definition 5.6. The operator $T_P$ is growing if for all interpretations $I$, $J$, and $M$, $I \preceq J \preceq M \preceq G(T_P \uparrow \omega(I))$ implies that $T_P(J) \subseteq T_P(M)$.

Consider Example 5.7 again. Let $I = \{\}$, $J$, $M$ and $G(T_P \uparrow \omega(\{\}))$ be as follows:

$J = \{edge(a, \{b\}), edge(b, \{c\}), edge(c, \{d, e\})\}$
$M = \{edge(a, \{b\}), edge(b, \{c\}), edge(c, \{d, e\}),$
$reach(a, \{b\}), reach(b, \{c\}), reach(c, \{d, e\})\}$
\[ G(T_P \uparrow \omega(\{\})) = \{\text{edge}(a, \{b\}), \text{edge}(b, \{c\}), \text{edge}(c, \{d, e\}), \text{reach}(a, \{b, c, d, e\}), \text{reach}(b, \{c, d, e\}), \text{reach}(c, \{d, e\})\} \]

Then \( I \preceq J \preceq M \preceq G(T_P \uparrow \omega(I)) \).

\[ T_P(J) = \{\text{edge}(a, \{b\}), \text{edge}(b, \{c\}), \text{edge}(c, \{d, e\}), \text{reach}(a, \{b\}), \text{reach}(b, \{c\}), \text{reach}(c, \{d, e\}), \text{reach}(c, \{d, e\})\} \]

\[ T_P(M) = \{\text{edge}(a, \{b\}), \text{edge}(b, \{c\}), \text{edge}(c, \{d, e\}), \text{reach}(a, \{b\}), \text{reach}(b, \{c\}), \text{reach}(c, \{d, e\}), \text{reach}(a, \{c\}), \text{reach}(b, \{d\}), \text{reach}(b, \{e\})\} \]

We have \( T_P(J) \subseteq T_P(M) \).

**Definition 5.7.** Let \( T_{P_1}, \ldots, T_{P_n} \) be a sequence of operators. The iterative powers of the sequence with respect to an interpretation \( M \) are defined by

\[
\begin{align*}
M_0 &= \{\}, \\
M_1 &= G(T_{P_1} \uparrow \omega(M_0)), & \text{if } G(T_{P_1} \uparrow \omega) \text{ is defined} \\
M_2 &= G(T_{P_2} \uparrow \omega(M_1)), & \text{if } G(T_{P_2} \uparrow \omega) \text{ is defined} \\
& \vdots \\
M_p &= M_n = G(T_{P_n} \uparrow \omega(M_{n-1})). & \text{if } G(T_{P_n} \uparrow \omega) \text{ is defined}
\end{align*}
\]

Let \( I \) be an interpretation, and \( D \) a set of predicate symbols. We denote by \( \pi_D(I) \) the extensions of predicate symbols of \( D \) in \( I \) defined by:

\[ \pi_D(I) = \{A \in I \mid \text{the predicate symbol of } A \text{ is in } D\} \]

**Lemma 5.1.** Let \( P \) be a program stratified by \( P = P_1 \cup \ldots \cup P_n \), \( D_1, \ldots, D_n \) the sets of predicate symbols of the corresponding strata, \( I \) an interpretation, and \( T_{P_1}, \ldots, T_{P_n} \) the corresponding sequence of operator of \( P \). Then \( \pi_{D_1 \cup \ldots \cup D_{j-1}}(I) = \pi_{D_1 \cup \ldots \cup D_{j-1}}(T_{P_j} \uparrow \omega(I)) \).

**Proof.** We prove by induction on \( i \) that \( \pi_{D_1 \cup \ldots \cup D_{j-1}}(I) = \pi_{D_1 \cup \ldots \cup D_{j-1}}(T_{P_j} \uparrow i(I)) \). The claim is obvious true for \( i = 0 \), since \( T_{P_j} \uparrow 0(I) = I \). Since the predicate symbols in the head of rules of \( P_j \) are not in \( D_1 \cup \ldots \cup D_{j-1} \). The application of \( T_{P_j} \) to \( T_{P_j} \uparrow i(I) \) does not change the extensions of the predicate symbols of \( D_1 \cup \ldots \cup D_{j-1} \).

**Corollary 5.1.** Let \( P \) be a program stratified by \( P = P_1 \cup \ldots \cup P_n \). Then for each \( i \) and \( j \) such that \( 1 \leq i \leq j \leq n \), we have

1. \( \pi_{D_1 \cup \ldots \cup D_i}(M_i) = \pi_{D_1 \cup \ldots \cup D_j}(M_j) \);
2. \( M_i \preceq M_j \);
3. \( T_{P_i}(M_i) = T_{P_j}(M_j) \).

**Proof.** Straightforward from Lemma 5.1. □

**Lemma 5.2.** Let \( P \) be a well-typed program stratified by \( P = P_1 \cup \ldots \cup P_n \), \( T_{P_1}, \ldots, T_{P_n} \) the corresponding sequence of operator of \( P \). Then \( T_{P_i} \) is growing for \( 1 \leq i \leq n \).

**Proof.** Let \( I \preceq J \preceq M \preceq G(T_{P_i} \uparrow \omega(I)) \) and \( A \in T_{P_i}(J) \). Then there is an instantiated rule in \( P_i \) of the form \( A \vdash L_1, \ldots, L_n \) such that \( J \models L_1, \ldots, J \models L_n \). For each \( j \), if \( L_j \) is negative or contains a complete set, then the predicate symbols
of $L_j$ is in $D_1 \cup \ldots \cup D_{i-1}$. By Lemma 5.1, $\pi_{D_1 \cup \ldots \cup D_{i-1}}(J) = \pi_{D_1 \cup \ldots \cup D_{i-1}}(M)$, so $M \models L_j$. Otherwise, $L_j$ is a non-negated atom that does not contain any complete sets. There are two possibilities: $L_j \in J$ or $L_j \notin J$. Consider first $L_j \in J$. As $J \preceq M$, there exists $B \in M$ such that $L_i \preceq B$. By Proposition 4.4 (1), $L_j = B$. Thus, $M \models L_j$. If $L_j \notin J$, then there is a fact $B \in J$ such that $L_j \preceq B$ by Definition 11. Since $J \preceq M$, there exists a $B' \in M$ such that $B \preceq B'$. By Proposition 4.4 (2), $L_j \preceq B'$. Thus $M \models L_j$. Therefore $A \in TP(M)$. □

Part of the following claim has been proved in [19]. The proof for Relationlog is essentially the same. It is reproduced here for convenience.

**Theorem 5.1.** Let $P$ be a well-typed program stratified by $P = P_1 \cup \ldots \cup P_n, D_1, \ldots, D_n$ the sets of predicate symbols of the corresponding strata, and $TP_1, \ldots, TP_n$ the corresponding sequence of operators of $P$. Then:

1. $M_P$ is a model of $P$.
2. $M_P$ is a minimal model of $P$.
3. $M_P$ is a supported model of $P$.
4. $M_P$ is the (unique) perfect model of $P$.

**Proof.** (1) Let $P_0$ be an empty set. We show, using induction on $i$, that $M_i$ is a model of $P_0 \cup \ldots \cup P_i$. When $i = n$, $M_P = M_n$ is then a model of $P$.

For the basis, $M_0 = \{\}$ is obviously a model of $P_0$. Assume the claim holds for some $i \geq 0$. We now prove $M_{i+1}$ is a model of $P_0 \cup \ldots \cup P_{i+1}$. If $M_{i+1}$ is not a model, then there are two possibilities by Definition 11. One is that there exists a rule $r \in P_0 \cup \ldots \cup P_{i+1}$ of the form $A \leftarrow L_1, \ldots, L_n$ such that $M_{i+1} \models \lnot r$. If $r \in P_0 \cup \ldots \cup P_i$, then the predicate symbols of $A, L_1, \ldots, L_n$ are in $D_1 \cup \ldots \cup D_i$. Since $\pi_{D_1 \cup \ldots \cup D_i}(M_i) = \pi_{D_1 \cup \ldots \cup D_i}(M_{i+1})$ by Corollary 5.1 (1) and $M_i$ is a model of $P_0 \cup \ldots \cup P_i$, $M_{i+1} \models r$, which is a contradiction. If $r \in P_{i+1}$, and $M_{i+1} \models \lnot r$, then there exists a ground substitution $\theta$, such that $M_{i+1} \models \theta L_1, \ldots, M_{i+1} \models \theta L_n$, and $M_{i+1} \not\models \theta A$. For each $j$, if $\theta L_j$ is negative or has a complete set, then the predicate symbols of $L_j$ is in $D_1 \cup \ldots \cup D_i$. So $G(TP_{i+1} \uparrow k(M_i)) \models \theta L_j$ for all $k \geq 0$. For this $j$, set $\alpha(j) = 0$. Otherwise, there exists a $k$ such that $G(TP_{i+1} \uparrow k(M_i)) \models \theta L_j$. Let $\alpha(j)$ denote this $k$ and $l$ denote $\max\{\alpha(j) \mid 1 \leq j \leq n\}$. By Lemma 5.2, $TP_{i+1}$, is growing. By Proposition 5.1 (6), for $l' > l$, $G(TP_{i+1} \uparrow l'(M_i)) \models \theta A$. Thus, $M_{i+1} = G(TP_{i+1} \uparrow \omega(M_i)) \models \theta A$, which is again a contradiction. So, for each rule $r \in P_0 \cup \ldots \cup P_{i+1}$, $M_{i+1} \models r$.

The other case to be considered is that for all instantiated rules satisfied by $M_{i+1}$ such that their heads are compatible, there does not exist a fact in $M_{i+1}$ in such that each head is part-of. Let $S$ be such a set of instantiated rules and $S_h$ be the set of their heads. Then $S_h$ is a set of facts with the same predicate symbol $p$. If $p \in D_h \cup \ldots \cup D_n$, then there exists a fact $A$ in $M_i$ such that $S_h \models A$ as $M_i$ is a model. Since $\pi_{D_h \cup \ldots \cup D_n}(M_i) = \pi_{D_h \cup \ldots \cup D_n}(M_{i+1})$, $A \in M_{i+1}$, which is a contradiction. If $p \in D_{i+1}$, then there exists a $k$ such that $S_h \subseteq TP_{i+1} \uparrow k(M_i) \subseteq TP \uparrow \omega(M_i)$. As $S_h$ is the maximal compatible set of $TP \uparrow \omega(M_i)$, $G(S_h) = \{A\} \subseteq G(TP \uparrow \omega(M_i)) = M_{i+1}$. Thus for the given $S_h$, there exists $A \in M_{i+1}$ such that $S_h \models A$, which is again a contradiction. Therefore, $M_{i+1}$ is a model of $P_0 \cup \ldots \cup P_{i+1}$.

(2) Let $N$ be a model of $P$. We prove by induction on $i$ that
if $N \preceq M_i$, then $M_i \preceq N$.  \hfill (1)

The basis clearly holds.  Assume the claim holds for $i \geq 0$.  We prove by induction on $j$ that

$$G(T_{P_{i+1}} \uparrow j(M_i)) \preceq N.$$  \hfill (2)

For $j = 0$, it is true by hypothesis.  Suppose it is true for $j \geq 0$.  Since $N \preceq M_i$ and $T_{P_{i+1}}$ is growing, $T_{P_{i+1}}(G(T_{P_{i+1}} \uparrow j(M_i))) \preceq T_{P_{i+1}}(N)$ by Lemma 5.2.  We now prove by induction on $k$ that

if $N$ contains $M_i$, then $T_{P_{i+1}} \uparrow k(M_i) \subseteq M_i \cup T_{P_{i+1}}(N)$  \hfill (3)

The claim is clearly true for $k = 0$.  Assume the claim holds for $k \geq 0$.  Then

$$T_{P_{i+1}} \uparrow k(M_i) \subseteq M_i \cup T_{P_{i+1}}(N)$$  \hfill by induction hypothesis

$$G(T_{P_{i+1}} \uparrow k(M_i)) \preceq G(M_i \cup T_{P_{i+1}}(N))$$  \hfill by Proposition 5.1 (6)

$$= M_i \cup G(T_{P_{i+1}}(N))$$  \hfill by Proposition 5.1 (5) and (4)

$$\preceq N$$  \hfill $N$ is a model & by Proposition 5.2

$$T_{P_{i+1}}(G(T_{P_{i+1}} \uparrow k(M_i))) \subseteq T_{P_{i+1}}(N)$$  \hfill by Lemma 5.2 and $N \preceq M_i$

$$T_{P_{i+1}} \uparrow (k + 1)(M_i) = T_{P_{i+1}}(G(T_{P_{i+1}} \uparrow k(M_i))) \cup T_{P_{i+1}} \uparrow k(M_i)$$

$$\subseteq M_i \cup T_{P_{i+1}}(N)$$  \hfill by induction hypothesis

Thus (3) holds for all $k$.

$$T_{P_{i+1}} \uparrow (j + 1)(M_i) = T_{P_{i+1}}(G(T_{P_{i+1}} \uparrow j(M_i))) \cup T_{P_{i+1}} \uparrow j(M_i)$$

$$\subseteq M_i \cup T_{P_{i+1}}(N)$$  \hfill by (4) and (3)

$$G(T_{P_{i+1}} \uparrow (j + 1)(M_i)) \preceq G(M_i \cup T_{P_{i+1}}(N))$$  \hfill by Proposition 5.1 (6)

$$= M_i \cup G(T_{P_{i+1}}(N))$$  \hfill by Proposition 5.1 (5) and (4)

$$\preceq N$$  \hfill by assumption and Proposition 5.2

So (2) holds for all $j$, we have $M_{i+1} \preceq N$.  This proves (1).

(3) Let $P_0$ be an empty set.  Now we prove by induction on $i$ that $M_i$ is a supported model of $P_0 \cup \ldots \cup P_i$ by showing $M_i \preceq G(T_{P_i}(M_i))$ where $T_{P_i}(I) = \cup_{j=0}^i T_{P_j}(I)$.  When $i = n$, $M_n$ is then a supported model of $P$ by definition.

The basis is clearly true.  Assume the claim holds for $i \geq 0$.  Let $M_{i+1} = \{\}$.  We now prove by induction for all $j \geq 0$

$$M_j \preceq M_{j-1} \cup G(T_{P_j}(M_j)).$$  \hfill (5)

The basis is clearly true.  Assume the claim holds for $j \geq 0$.  In order to prove that (5) is true for $j + 1$, we first prove by induction on $k$ that

$$T_{P_{j+1}} \uparrow k(M_j) \subseteq M_j \cup T_{P_{j+1}}(M_{j+1})$$  \hfill (6)

The basis is clearly true.  Assume the claim holds for $k \geq 0$.  Then
\[ T_{P_{j+1}^i} \uparrow k(M_j) \subseteq T_{P_{j+1}^i} \uparrow \omega(M_j) \quad \text{by definition} \]
\[ G(T_{P_{j+1}^i} \uparrow k(M_j)) \subseteq G(T_{P_{j+1}^i} \uparrow \omega(M_j)) \quad \text{by Proposition 5.1 (7)} \]
\[ M_j^{i+1} = T_{P_{j+1}^i} \uparrow k(M_j) = M_{j+1} \quad \text{by definition} \]
\[ T_{P_{j+1}^i}(G(T_{P_{j+1}^i} \uparrow k(M_j))) \subseteq T_{P_{j+1}^i}(M_{j+1}) \quad \text{by Lemma 5.2 (7)} \]
\[ T_{P_{j+1}^i} \uparrow (k + 1)(M_j) = T_{P_{j+1}^i}(G(T_{P_{j+1}^i} \uparrow k(M_j))) \cup T_{P_{j+1}^i} \uparrow k(M_j) \subseteq M_j \cup T_{P_{j+1}^i}(M_{j+1}) \quad \text{by (7) and induction hypothesis} \]

Thus (6) holds for all \( k \). Consequently

\[ T_{P_{j+1}^i} \uparrow \omega(M_j) \subseteq M_j \cup T_{P_{j+1}^i}(M_{j+1}) \quad \text{by (6)} \]
\[ M_j^{i+1} = G(T_{P_{j+1}^i}(M_j)) = G(M_j \cup T_{P_{j+1}^i}(M_{j+1})) \quad \text{by definition} \]
\[ \leq M_j \cup G(T_{P_{j+1}^i}(M_{j+1})) \quad \text{by Proposition 5.1 (6) and (5)} \]

So (5) holds for all \( j \geq 1 \). Now consider \( i \).

\[ M_i \preceq M_{i-1} \cup G(T_{P_i}(M_i)) \quad \text{by (5)} \]
\[ \leq G(T_{P_{i-1}^j}(M_{i-1})) \cup G(T_{P_i}(M_i)) \quad \text{by induction hypothesis} \]
\[ = G(T_{P_{i-1}^j}(M_{i-1})) \cup T_{P_i}(M_i) \quad \text{by Proposition 5.1 (6)} \]
\[ \leq G(T_{P_{i-1}^j}(M_i) \cup T_{P_i}(M_i)) \quad \text{by Corollary 5.1 (3)} \]
\[ = G(T_{P_i}(M_i)) \quad \text{by definition} \]

which concludes the proof.

(4). Let \( N \) be a model of \( P \) that is different from \( M_P \). Then \( N_i = P_{D_6, \ldots, D_i}(N) \) is a model of \( P_0 \cup \ldots P_i \). Let \( j \) be the smallest integer such that \( M_j \neq N_j \). Then \( M_i = N_i \) for all \( i < j \). From the proof of (2), \( M_j \preceq N_j \). Let \( A \in M_P - N \), then either \( A \in M_j - N \) or \( A \in M_k - N \) for some \( k > j \). If \( A \in M_j - N \), then \( A \in M_j - N_j \) and there exists an \( A' \in N_j - M_j \) as well as \( A' \subset N - M_P \) such that \( A \preceq A' \) since \( M_j \preceq N_j \). If \( A \in M_k - N \) for some \( k > j \), then there exists \( A' \in N_j - M_P \) since \( M_j \preceq N_j \) and \( M_j \) and \( N_j \) are different. Therefore, \( M_P \) is preferable to \( N \) by Definition 17. \( \square \)

6. CONCLUSION

In this paper, we have presented a logic programming based language Relationlog for nest relational and complex value models. It is a typed extension to Datalog with sets and tuples with a pure declarative semantics that captures the intended semantics of nested sets, tuples and relations, and also a bottom-up fixpoint semantics which coincides with its declarative semantics. The main novel feature of the language is the powerful mechanism to represent and manipulate partial and complete information on nested sets, tuples and relations, which generalize the set grouping and set enumeration mechanisms of LDL. They allow direct inference and access to deeply embedded values so that extended relational algebra operations as defined in [1, 14, 26] can be represented directly, and more importantly, recursively in a way similar to Datalog.

There are several open issues which we still need to address. We intend to investigate how to incorporate update constructs into the language to make it a complete database language. Besides, we would also like to investigate how to support unknown values that are common in database applications. Indeed, it is not clear how to perform deduction when there are unknown values in a deductive
database. We are currently investigating how to efficiently implement the language based on the techniques used in LDL [12], CORAL [24] and Atlas [27].

The author wishes to thank the anonymous referees for their detailed comments and suggestions which have significantly improved the quality and accuracy of the paper. This work was supported by the Natural Sciences and Engineering Research Council of Canada.

REFERENCES


