A Typed Deductive and Object-Oriented Database Language

Mengchi Liu

Technical Report CS-96-03
March, 1996

Department of Computer Science
University of Regina
Regina, Saskatchewan, CANADA
S4S 0A2

ISSN 0828-3494
ISBN 0-7731-0317-1

A Typed Deductive and Object-Oriented Database Language

Mengchi Liu

Department of Computer Science
University of Regina
Regina, Saskatchewan
Canada S4S 0A2

Fax: (306) 585-4700
Email: mliu@cs.uregina.ca

Abstract

This paper presents a novel typed deductive object-oriented database language, called ROL (Rule-based Object Language), which is being developed at the University of Regana. This language is a declarative language. It can naturally and directly support object-oriented features such as object identity, complex objects, classes, class hierarchy, multiple inheritance with overriding, and schema in a deductive framework. It also treats sets and relationships as first class citizens and provides powerful mechanisms for representing both partial and complete information on sets. Another novelty is the introduction of a new ordering which can capture the intended semantics of nested sets. The ROL language is given a logical semantics that cleanly accounts for object-oriented features and the usage of schema in a deductive framework.

Keywords: Deductive databases, object-oriented databases, minimal model semantics, fixpoint semantics.

1 Introduction

Deductive and object-oriented databases are two important extensions of the traditional database technology. Deductive databases extend the expressive power of traditional databases by means of recursion and declarative querying with a firm logical foundation. Examples of such languages are Datalog [18], LDL [33] and CORAL [35]. Object-oriented databases extend the data modeling power of the traditional databases by means of object identity, complex objects, classes, class hierarchy, and inheritance. Examples of such languages are Iris [22], Exodus [17], Orion [27], O_2 [28] and Jasmine [24]. However, both extensions have shortcomings. Deductive databases lack powerful data modeling mechanisms, while object-oriented databases lack logical semantics and declarative query languages. In the past few years, a lot of efforts have been made to integrate deductive and object-oriented databases to gain the best of the two approaches. A number of deductive object-oriented database languages have been proposed, such as O-logic [32], revised O-logic [26], C-logic [19], IQL [5], IQL2[1], F-logic [25], LOGRES [16], LLO [31], LOL [14], Datalog$^{\text{method}}$ [6], DLT [10], Gulog [21] and Rock & Roll [12].

However, none of these proposals have achieved the objectives of providing the necessary expressive power for database applications in a declarative fashion and providing logical semantics for object-oriented features for the following reasons.

Sets play an important role in object-oriented data models. However, sets are not treated as first-class citizens, as an object can have a set as an attribute value but a set cannot have any attributes. Most deductive object-oriented database language proposals inherit this property, which significantly restricts their expressive power. In [9], several tasks for a manufacturing company's parts database are proposed to test database languages. One task is to obtain the assembly cost and mass increment of a composite part. Because of inadequate support of sets, none of the proposed deductive object-oriented languages are powerful enough to perform this task declaratively, including the most powerful language F-logic. In languages such as Rock & Roll [12] and IQL2 [1], this task can be done, but only by resorting to a low-level imperative program rather than a high-level declarative program.

On the other hand, powerful set representation mechanisms have been proposed in value-oriented deductive database languages such as LDL [13], CORAL [35] and COL [2]. In LDL, for example, sets can be used either in enumerated form or as a result of element grouping, set-valued variables can be used to range over sets, and sets can contain sets. As a result, the above database task can be performed declaratively in LDL.

Besides, in deductive object-oriented database languages that support sets as attribute values, such as revised O-logic or F-logic, set information can only be represented in a limited way. For example, we can only say that *ann* is one of the parents of *tom* with a singleton set $\{ann\}$. We cannot directly say that *ann* is the only parent of *tom* by any means. The complete information for the *parents* attribute of *tom* depends on what else we have in the program. Similarly, we cannot directly say that *tom* is an orphan with an empty set of parents, as doing so actually means something different: *tom*'s parents information is known, rather than unknown. In other words, we can only specify the partial information, rather than the complete information of a set-valued attribute. This special set treatment does have its merits, as more programs can be stratified with it than in other languages without it such as LDL and COL.

Finally, a database usually has a schema. The schema provides the description of the database structure and also constrains data in the database. It corresponds to the type declarations of

programming languages and is the basis for storage structure and query optimization strategies. However, the usage of the schema is not considered in most deductive database languages, because their semantics are based on the work on the logic programming language Prolog which is not typed. As a result, the semantics of schema, especially how the schema could constrain the data in the database, is not well-defined in most deductive frameworks, value-oriented or object-oriented. Some of the deductive object-oriented database languages, such as Rock & Roll, even have no logical semantics. However, object-oriented features such as complex object structures, class hierarchy, multiple inheritance with overriding are mainly schematic features.

In this paper, we propose a novel deductive object-oriented language called ROL (Rule-based Object Language). This language has both the necessary expressive power and data modeling power for advanced database applications. It effectively integrates important features in deductive databases and object-oriented databases into a uniform framework. It has a simple, easy to use, but powerful syntax compared to other proposals.

It treats sets as first-class citizens and provides powerful set representation mechanisms that integrates the set treatments of both LDL and F-logic . In order to do so, we introduce a new operator called the compaction operator. The database task mentioned above can be handled in ROL declaratively as in LDL, and more importantly object-orientedly, (see Example 2.8). By also supporting the set treatment of F-logic, more programs can be stratified in ROL than in languages such as LDL and COL.

ROL also treats relationships as first-class citizens and allows relationships to be treated as objects and to participate in other relationships. As a result, Datalog and LDL without grouping are simply special cases, (see Example 2.7). None of the reported deductive object-oriented database languages can do this. LOGRES and IQL2 in this regard are close to ROL. They both support classes and relations. However, they are strictly based on the Entity/Relationship model and don't allow relationships to be used as attribute values of objects or participate in other relationships. Indeed, the user of ROL can choose either the value-oriented approach, the object-oriented approach, or their combination.

ROL directly supports the notion of schema and is a typed language without using typed symbols such as typed variables in programs and queries syntactically as in [30, 34], COL [2], Gulog [21]. Instead, symbols of a ROL program are bound to certain types (called classes) based on the schema in the intended semantics of the program.

Most importantly, ROL has a logical semantics that cleanly accounts for object-oriented features such as object identity, complex objects, classes, class hierarchy, multiple inheritance with overriding, especially the usage of schema in a deductive framework. As a result, the semantics of ROL differs significantly from that of untyped languages such as Prolog, Datalog [18], LDL, F-logic (and its predecessors), and also typed languages such as those in [30, 34], COL, Gulog.

Another novelty of this paper is the introduction of a new ordering which is able to capture the intended semantics of nested sets. ROL is given Herbrand minimal model semantics based on this new ordering. A stratification in the spirit of several other researchers [2, 8, 13] is used. We follow the standard treatment of logic programming languages and show that for a stratified program, the unique minimal and supported model, when it exists, can be computed bottom-up using a finite sequence of fixpoints and used as the intended semantics of the program.

The rest of the paper is organized as follows. Section 2 introduces the syntax of ROL. Section 3 presents the Herbrand model semantics of ROL. Section 4 discusses how to compare different

interpretations and models, and introduces the notion of minimal model. Section 5 focuses on the stratification restriction on programs and their semantics. Section 6 concludes the paper.

2 Syntax of ROL

This section defines the formal syntax of the ROL language. We assume the existence of the following pairwise disjoint sets:

(1) the set of value class names $\mathcal{B} = \{integer, string\}$;
(2) a countably infinite set $\mathcal{C}$ of object class names which are function symbols with arity $n \geq 0$;
(3) a countably infinite set $\mathcal{A}$ of attribute labels;
(4) a countably infinite set $\mathcal{D}$ of values which is the union of the set $\mathcal{I}$ of integers and the set $\mathcal{S}$ of strings;
(5) a countably infinite set $\mathcal{O}$ of object identifiers;
(6) a countably infinite set $\mathcal{V}$ of variables.

A ROL database consists of two parts: a schema and a program. The schema contains information about classes, while the program contains information about objects. The schema is used to constrain the objects and their attribute values that can be generated in the program. We first define schema.

Definition 2.1 The *classes* are defined as follows:

(1) *integer* is a *value* class.
(2) Let $i_1 \in \mathcal{I}$, $i_2 \in \mathcal{I}$ and $i_1 \leq i_2$. Then $integer(i_1..i_2)$ is a *value* class.
(3) *string* is a *value* class.
(4) Let $s_i \in \mathcal{S}$ for $i = 1, ..., n$. Then $string(\{s_1, ..., s_n\})$ is a *value* class.
(5) If $p_1, ..., p_n$ are classes and f is an n-ary object class name, then $f(p_1, ..., p_n)$ is an *object* class. If $n = 0$, f is called an *atomic* object class. Otherwise, it is called a *constructed* object class.
(6) If p is a class, then $\{p\}$ is a *set* class; p is the *component* class of $\{p\}$.

Classes in ROL are used to denote collections of objects that share common attributes. Based on the definition, three kinds of classes are distinguished: value classes, object classes and set classes. Object classes are classified into two kinds: atomic object classes and constructed object classes. The collections which value classes denote are fixed in ROL. In details, *integer*, $integer(i_1..i_2)$, *string*, and $string(\{s_1, ..., s_n\})$ denote $\mathcal{I}$, $\{i \mid i_1 \leq i \leq i_2\}$, $\mathcal{S}$, and $\{s_1, ..., s_n\}$ respectively. The collections which object classes denote depend on the ROL program. The collections which set classes denote depend on the collections which the component classes denote.

Example 2.1 Following are examples of classes:

Value classes	$integer, string, integer(15..30), string(\{\text{“}Male\text{”}, \text{“}Female\text{”}\})$
Atomic object classes	$person$, $student$, $employee$, $dept$, $course$, $part$, $supplier$
Constructed object classes	$family(\{person\})$, $supplies(supplier, part)$
Set classes	$\{integer\}, \{person\}, \{family(\{person\})\}$

Definition 2.2 If p and q are object classes, then p *isa* q defines p to be an *immediate subclass* of q and q to be an *immediate superclass* of p.

For example, we may use *student isa person* and $family(\{person\})$ *isa taxunit* to define that *student* is an immediate subclass of *person* and $family(\{person\})$ is an immediate subclass of *taxunit* respectively.

Definition 2.3 If p and q are classes and l is an attribute label, then $p[l \Rightarrow q]$ defines an *attribute* l of p. If q is a set class, then l is called a *set-valued* attribute. Otherwise, l is called a *single-valued* attribute.

The attribute definition $p[l \Rightarrow q]$ actually specifies p as a key in the sense of relational databases which single-valued (i.e., functionally) or multi-valued determines q through l. For example, the classical functional dependency of the suppliers-parts database in [20] can be represented in ROL with $supplies(supplier, part)[quantity \Rightarrow integer]$. Such dependencies are built into the semantics of ROL by the consistency constraint. Furthermore, the referential integrity rule is also built into the semantics of ROL by the typing constraint. Both notions will be introduced in the next section.

We use $p[l_1 \Rightarrow q_1, ..., l_n \Rightarrow q_n]$ *isa* $p_1, ..., p_m$ to stand for $p[l_1 \Rightarrow q_1]$, ..., $p[l_n \Rightarrow q_n]$, p *isa* p_1, ..., p *isa* p_m.

Definition 2.4 A *schema* is a tuple $K = (B, C, A, D_C, isa, D_A)$, where

(1) $B \subseteq \mathcal{B}$ is a finite set of value class names;
(2) $C \subseteq \mathcal{C}$ is a finite set of object class names;
(3) $A \subseteq \mathcal{A}$ is a finite set of attribute labels;
(4) D_C is a finite set of class definitions based on B and C such that every object class name $f \in C$ is associated with exactly one class in D_C;
(5) isa is a finite set of immediate subclass definitions;
(6) D_A is a finite set of attribute definitions such that all classes and attribute labels used in D_A are in $D_C \cup A$.

The above definition for schema is an abstract representation. In the following examples, we use an intuitive representation. The conversion from the abstract one to the intuitive one is straightforward.

It is our intention that a subclass inherits all the attribute definitions from its superclasses but may override or refine some of the inherited attribute definitions and introduce extra attribute definitions local to itself.

Example 2.2 The following is a schema of ROL.

$person[age \Rightarrow integer(1..125), father \Rightarrow person, mother \Rightarrow person,$
$\quad parents \Rightarrow \{person\}, ancestors \Rightarrow \{person\}, trueancestors \Rightarrow \{person\}]$
$student[age \Rightarrow integer(15..30), taking \Rightarrow \{course\}]$ *isa person*
$employee[age \Rightarrow integer(18..65), salary \Rightarrow integer(0..10000)]$ *isa person*
workstudent isa employee, student
$course[name \Rightarrow string(\{\text{“}Databases\text{”}, \text{“}Logic\text{”}, \text{“}Programming\text{”}\})]$

$taxunit[income \Rightarrow integer]$
$family(\{person\})[children \Rightarrow \{person\}, childrencount \Rightarrow integer]$ *isa* $taxunit$
$\{person\}[count \Rightarrow integer]$

The class *person* has single-valued attributes *age*, $father$, and *mother*, and set-valued attributes *parents*, *ancestors* and *trueancestors*. The class *student* is an immediate subclass of *person* which refines the attribute *age* and introduces a set-valued attribute *takes*. The class *employee* is an immediate subclass of *person* which also refines the attribute *age* and introduces a single-valued attribute *salary*. The class *workstudent* is an immediate subclass of *student* and *employee*. The class *course* has a single-valued attribute *name*. The class *taxunit* has a single-valued attribute *income*. The constructed object class $family(\{person\})$ has a set-valued attribute *children* and a single-valued attribute *childrencount*. The set class $\{person\}$ has a set-valued attribute *count*.

We now extend *isa* and D_A to capture our intention as follows.

Definition 2.5 For a given schema $K = (B, C, A, D_C, isa, D_A)$, *extended-isa* (abbreviated by *e-isa*) is a binary relationship, which satisfies the following:

(1) p *e-isa* q if both p and q are value classes and the collection of values denoted by p is a subset of the collection of values denoted by q;
(2) p *e-isa* q if p *isa* q;
(3) $\{p\}$ *e-isa* $\{q\}$ if p *e-isa* q.

We note isa^* and *e-isa** the reflexive and transitive closures of *isa* and *e-isa* respectively.

The isa^* relationship captures the subclass relationship among object classes, while *e-isa** captures the subclass relationship among all classes. Based on Example 2.2, we have

$person$ *e-isa** $person$
$workstudent$ *e-isa** $person$
$family(\{person\})$ *e-isa** $taxunit$
$integer(15..30)$ *e-isa** $integer(1..125)$
$integer(15..30)$ *e-isa** $integer$
$\{student\}$ *e-isa** $\{person\}$
$\{workstudent\}$ *e-isa** $\{person\}$
...

Note that the subclass relationship is not extended to constructed object classes. Our intention is to treat a constructed object class $f(p_1, ..., p_n)$ as a unique class associated with the class name f. For example, $family(\{person\})$ is the only class associated with $family$, we cannot have $family(\{student\})$, $family(\{employee\})$.

Definition 2.6 A schema $K = (B, C, A, D_C, isa, D_A)$ is *well-defined* iff there does not exist distinct p and q such that p *e-isa** q and q *e-isa** p.

The problem with circular subclass relationship has been addressed in [3].

Proposition 2.1 If the schema is well-defined, then the *e-isa** relationship is a partial order over all classes.

Proof. Straightforward from Definition 2.6. □

Note that the well-definedness requirement is less restrictive than the requirement that all classes form a lattice such as in LOGIN [7] and LIVING IN A LATTICE [23].

From now on, we use schema to refer to well-defined schema.

We next introduce the following notion to capture the multiple inheritance with overriding.

$$D_A^* = \{p[l \Rightarrow q'] \mid \exists p', p \ \textit{e-isa}^* \ p', p'[l \Rightarrow q'] \in D_A, \text{ and not } \exists p'', p' \neq p'', \\ p \ \textit{e-isa}^* \ p'', p'' \ \textit{e-isa}^* \ p', p''[l \Rightarrow q''] \in D_A\}$$

Proposition 2.2 Let $K = (B, C, A, D_C, isa, D_A)$ be a well-defined schema. Then the notion D_A^* has the following properties:

(1) $D_A \subseteq D_A^*$.
(2) If p_0 *e-isa** p_1, p_1 *e-isa** p_2, $p_1 \neq p_2$, $p_1[l \Rightarrow q_1] \in D_A$, $p_2[l \Rightarrow q_2] \in D_A$, and there does not exist p_1' such that p_0 *e-isa** p_1', p_1' *e-isa** p_1, $p_1' \neq p_1$, and $p_1'[l \Rightarrow q_1']$, then $p_0[l \Rightarrow q_1] \in D_A^*$ and $p_0[l \Rightarrow q_2] \notin D_A^*$.
(3) If p_0 *e-isa** p_1, $p_0 \neq p_1$, $p_0[l \Rightarrow q_0] \in D_A$, and $p_1[l \Rightarrow q_1] \in D_A$, then $p_0[l \Rightarrow q_0] \in D_A^*$ and $p_0[l \Rightarrow q_1] \notin D_A^*$.

The first property says that D_A^* is an extension of D_A if the schema is well-defined. The second says that if both superclasses p_1 and p_2 of p_0 have attribute definitions for l, p_1 is the closer than p_2, and there does not exist such a class which is closer than p_1 (including p_0 itself), then p_0 inherits the attribute definition only from p_1 rather than p_2. As p_0 may have several such p_1s, p_0 therefore multiply inherits attribute definitions from these p_1s. The last says that if p_0 itself has an attribute definition for l, then p_0 uses this one rather than inherits from its superclasses. That is, p_0 overrides the attribute definition of its superclasses.

Proof. (1) Let $p[l \Rightarrow q] \in D_A$ and $p' = p$. Then we have $p'[l \Rightarrow q] \in D_A$. Since the schema is well-typed, we cannot find p'' satisfying the condition in the definition of D_A^*. Therefore $p[l \Rightarrow q] \in D_A^*$.

(2) Let $p = p_0, p' = p_1, p'' = p_1'$ in the definition of D_A^*. Then $p_0[l \Rightarrow q_1] \in D_A^*$. Let $p = p_0, p' = p_2, p'' = p_1$ in the definition of D_A^*. Then $p_0[l \Rightarrow q_2] \notin D_A^*$.

(3) Let $p = p_0, p' = p_1, p'' = p_0$ in the definition of D_A^*. Then $p_0[l \Rightarrow q_1] \notin D_A^*$. By (1), we have $p_0[l \Rightarrow q_0] \in D_A^*$. □

By (2) of Proposition 2.2, a subclass multiply inherits attribute definitions from its superclasses unless it overrides them. As a result, a class may have more than one attribute definition for the same attribute. That is, we may have $p[l \Rightarrow q_1]$ and $p[l \Rightarrow q_2]$ such that $q_1 \neq q_2$. They all constrain the attribute values of its instances. For the class *workstudent* in Example 2.2, the attribute *age* inherited from *student* and *employee* is constrained by both *integer*(15..30) and *integer*(18..65) which are equivalent to *integer*(18..30). In other words, constraints on a subclass are the conjunction of constraints on its superclasses.

By (3) of Proposition 2.2, in order for a subclass to override an attribute definition in ROL, we just need to introduce an attribute definition with the same name as the one inherited. For

example, *student* and *employee* override the attribute definition of *age* of *person* and inherit other attribute definitions, based on Example 2.2. By supporting overriding, we provide a means for dealing with conflicts of inherited attributes.

We next define terms and objects in ROL. Corresponding to classes, there are three kinds of terms in ROL: value terms, object terms and set terms. Object terms are divided into atomic object terms and constructed object terms. Set terms are divided into partial set terms and complete set terms.

Definition 2.7 A *term* is defined recursively as follows:

(1) a variable is either a *value* term, an *object* term, or a *complete* set term depending on the context;
(2) a value is a *value* term;
(3) an object identifier is an *atomic* object term;
(4) if f is an n-ary object class name from $\mathcal{C}$, and $O_1, ..., O_n, (n \geq 1)$ are terms other than partial set terms, then $f(O_1, ..., O_n)$ is a *constructed* object term;
(5) if $O_1, ..., O_n, (n \geq 1)$ are terms other than partial set terms, then $\langle O_1, ..., O_n \rangle$ is a *partial* set term;
(6) if $O_1, ..., O_n, (n \geq 0)$ are terms other than partial set terms, then $\{O_1, ..., O_n\}$ is a *complete* set term.

A term is *ground* if it has no variables. An *object* is a ground term. As terms, three kinds of objects are distinguished in ROL: values, *abstract objects* and *sets*. Abstract objects are divided into object identifiers and *constructed objects* which are ground constructed object terms. Sets are divided into *partial sets* which are ground partial set terms, and *complete sets* which are ground complete set terms. An object is *compact* if it isn't a partial set.

Example 2.3 Following are examples of legal objects:

Values:	5, 30, "*John*", "*Smith*"
Object identifiers:	$john, smith, mary$
Constructed objects:	$family(\{john, mary\}), g(family(\{john, mary\}), 1)$
Partial sets:	$\langle john, mary \rangle$, $\langle \{smith\}, \{smith, mary\} \rangle$
Complete sets:	$\{john, mary\}$, $\{\{smith\}, \{smith, mary\}\}$

However, $f(smith, \langle 2 \rangle), \{\langle john, mary \rangle\}$ and $\langle \langle john, mary \rangle \rangle$ are not objects in ROL.

It is our intention to use a partial set to denote part of a complete set. For example, given a complete set $\{smith, john\}$, we can denote part of this set by partial sets $\langle smith \rangle$, $\langle john \rangle$, and $\langle john, smith \rangle$. On the other hand, with these partial sets, we can obtain the corresponding complete set. We will discuss this issue in Section 5.2. As partial sets only represent part of complete sets, we do not allow variables to match them. So a variable is either a value term, an object term, or a complete set term, but not a partial set term. Given a partial set such as $\langle smith \rangle$, we cannot determine which complete set it denotes as it may denote $\{smith\}$, or $\{smith, john\}$. Because of their ambiguity, we do not allow partial set terms in constructed object terms and partial and complete set terms.

We will give ROL Herbrand model semantics in which values, object identifiers, and constructed objects are interpreted by themselves (i.e. uninterpreted). For example, 1 is interpreted

by 1, "$Smith$" by "$Smith$", $john$ by $john$, $family(\{john, mary\})$ by $family(\{john, mary\})$, and $supplies(s_1, p_1)$ by $supplies(s_1, p_1)$. The reason for this choice is that it is natural for database applications and is the way most logic programming and deductive database languages are given semantics. As a constructed object such as $supplies(s_1, p_1)$ is not a set based on such an interpretation, it is treated as a single-valued object. However, a constructed object with arity > 1 can be viewed naturally as a relationship between objects in ROL. For example, $supplies(s_1, p_1)$ can be treated as an object representing supplies and participates in other relationships and also as a relationship between s_1 and p_1.

Note that although the function symbols in constructed objects are uninterpreted, interpreted functions such as those used in COL [2] and [4] are also supported in ROL by means of attribute labels.

For convenience, we will use words starting with lower-case letters to denote ground terms, and words starting with capital letters to denote terms that may be nonground in the rest of the paper. In our discussion, we will treat partial sets and complete sets as sets in the traditional sense so that it makes sense to have both $b \in \langle a, b, c\rangle$ and $b \in \{a, b, c\}$.

Based on terms, we introduce object expressions whose counterparts in Datalog and Prolog are literals.

Definition 2.8 An *object expression* is defined as follows:

(1) if O is an object term and p is an object class, then $O : p$ is a *simple* object expression;
(2) if O is a constructed object term, then O is a *simple* object expression;
(3) if O is a term other than a partial set term, O' is a term, and l is an attribute, then $O[l \rightarrow O']$ is a *simple* object expression;
(4) if O, $O[l_1 \rightarrow O_1]$, ..., $O[l_n \rightarrow O_n]$ are simple object expressions, where $n \geq 0$, then $O[l_1 \rightarrow O_1, ..., l_n \rightarrow O_n]$ is an *composite* object expression, while O, $O[l_i \rightarrow O_i]$ for $i = 1, ..., n$ are called *constituent* object expressions;
(5) if $O : p$, $O[l_1 \rightarrow O_1]$, ..., $O[l_n \rightarrow O_n]$ are simple object expressions, where $n \geq 0$, then $O[l_1 \rightarrow O_1, ..., l_n \rightarrow O_n]$ and $O : p[l_1 \rightarrow O_1, ..., l_n \rightarrow O_n]$ are *composite* object expressions, while $O : p$, $O[l_i \rightarrow O_i]$ for $i = 1, ..., n$ are called *constituent* object expressions of the corresponding composite object expressions;
(6) if ψ is an object expression, then $\neg\psi$ is a negative object expression.

Note that a constructed object term itself is an object expression. As a result, all Datalog literals and LDL literals without set grouping are object expressions in ROL. Also note that simple and composite object expressions are not disjoint. For example, $f(S)[l \rightarrow Y]$ is both a simple and a composite object expression. As a composite object expression, it has constituent object expressions $f(S)$ and $f(S)[l \rightarrow Y]$.

An object expression is *ground* if it has no variables. A ground object expression is *compact* if it doesn't involve any partial sets. In a ground object expression $o[l \rightarrow o']$, o functions as a key which single-valued or set-valued determines o' as in the relational model, and it can be a value, an object identifier, a constructed object or a complete set. As in the relational model, we do not allow o (the key) to be a partial set as it is incomplete. Here l corresponds to data function of COL [2] and method of IQL2 [1] and F-logic [25].

In ROL, ground object expressions are used to assert instances of atomic object classes and constructed object classes and to specify attribute values of objects other than partial sets.

Example 2.4 Following are several ground object expressions:

$$smith : student[age \rightarrow 20, parents \rightarrow \{tom\}, grandparents \rightarrow \langle jack \rangle]$$
$$family(\{john, mary\})[children \rightarrow \{sam, tom\}, childrencount \rightarrow 2]$$
$$\{tom, sam\}[count \rightarrow 2]$$

The first one asserts that object identifier *smith* be an instance of *student* and specifies that *smith* has values 20 for attribute *age*, $\{tom\}$ for attribute *parents*, and $\langle jack \rangle$ for attribute *grandparents*. The complete set $\{tom\}$ means that it is the complete value for *parents*, while the partial set $\langle jack \rangle$ means that it is part of the value for *grandparents*. The second one specifies that the family of *john* and *mary* identified by $family(\{john, mary\})$ has children $\{sam, tom\}$ and the number of children is 2. The last specifies that the set $\{sam, tom\}$ identified by itself has 2 elements.

Consider the following examples of object expressions with variables.

$$smith : student[age \rightarrow X, parents \rightarrow Y]$$
$$family(S)[children \rightarrow X]$$
$$\{sam, tom\}[count \rightarrow Y]$$

Before we give the precise meanings to them, let us simply treat them as queries on the database containing the ground object expressions in Example 2.4 and briefly explain their meanings. The first one says list *smith*'s age X and his parents Y. Obviously X will match 20 and Y will match $\{tom\}$. The second says find the children X of the family of S. This time X will match $\{sam, tom\}$ and S will match $\{john, mary\}$. The last one says find the number Y of elements in the set $\{sam, tom\}$. Y will match 2. Note that we don't use any typed symbols syntactically so that X can match 20 in the first case and $\{sam, tom\}$ in the second case.

Arithmetic, set and comparison expressions in ROL are defined as in the standard arithmetic and set-theoretic theories. As partial sets cannot uniquely determine the corresponding complete sets, it doesn't make sense to operate on them. Therefore, we only allow complete set terms to be used in set and comparison expressions.

Example 2.5 Following are several ROL arithmetic, set and comparison expressions:

Arithmetic expressions:	$X + 10, X \times (Y - Z)$
Set expressions:	$\{john, smith\} \cup S_1,\ S_1 \cup S_2, (S_1 \cap S_2) \setminus S_3$
Comparison expressions:	$X = Y + Z, X + 10 > X \times (X - Z),$
	$X \in S, S = S_1 \cup S2, S_1 \cap S_2 = \{\}, (S_1 \cap S_2) \setminus S_3 \neq \{\}$

ROL is a rule-based language. We now introduce rules.

Definition 2.9 A *rule* is of the form A :- $L_1, ..., L_n$, where the head A is a non-negative object expression and the body $L_1, .., L_n, n \geq 0$ is a sequence of object and comparison expressions.

A *fact* is a rule with empty body, i.e., a ground object expression.

Rules of ROL are used to infer instances of atomic and constructed object classes and attribute values of objects other than partial sets. Instances of value classes such as *integer* and the corresponding operations over them are built-in so that we don't need to and cannot infer 20 :

integer using rules. Instances of set classes are automatically determined based on the instances of the corresponding component classes. For example, if the class *person* has instances *smith* and *mary*, then, $\{\}$, $\{smith\}$, $\{mary\}$ and $\{mary, smith\}$ are automatically instances of the set class $\{person\}$ and we cannot infer $\{smith\} : \{person\}$ using rules. The operations on sets are built-in as well. Instances of a subclass are also instances of its superclasses. We will discuss these issues in the section for semantics.

Example 2.6 Several ROL rules are given as follows:

$$
\begin{array}{l}
X[parents \rightarrow \langle Y \rangle] \text{ :- } X[father \rightarrow Y] \\
X[parents \rightarrow \langle Y \rangle] \text{ :- } X[mother \rightarrow Y] \\
X[ancestors \rightarrow \langle Y \rangle] \text{ :- } X[parents \rightarrow \langle Y \rangle] \\
X[ancestors \rightarrow \langle Y \rangle] \text{ :- } X[ancestors \rightarrow \langle Z \rangle], Z[parents \rightarrow \langle Y \rangle] \\
X[trueancestors \rightarrow \langle Y \rangle] \text{ :- } X[ancestors \rightarrow \langle Y \rangle], \neg X[parents \rightarrow \langle Y \rangle] \\
family(S)[children \rightarrow \langle X \rangle] \text{ :- } X[parents \rightarrow S] \\
X[childrencount \rightarrow Y] \text{ :- } X[children \rightarrow S], S[count \rightarrow Y] \\
\{X\}[count \rightarrow 1] \text{ :- } X : person \\
S[count \rightarrow X] \text{ :- } S_1[count \rightarrow Y], S_2[count \rightarrow Z], X = Y + Z, S = S_1 \cup S_2, S_1 \cap S_2 = \{\}
\end{array}
$$

The first two rules say that if Y is the father or the mother of X, then Y is one of the parents of X. The next two rules define the attribute *ancestors* of X recursively. The attribute *trueancestors* of X is defined using a negative object expression. The next rule involves a constructed object term. It says that if X has parents S (a set), then the family denoted by $family(S)$ has X as one of its children. The next rule says that if X has children S (a set) and S has Y members, then the *childrencount* of X is Y. The last two rules specify how to compute the number of elements in sets.

The above examples show how value terms, atomic object terms, constructed object terms, partial and complete set terms as well as set-valued variables can be used to infer information about objects in ROL. Especially, the last two rules show that traditional database aggregate operations such as *total*, *count*, etc. can be represented directly in ROL.

Note that all Datalog rules and LDL rules without set grouping are simply special cases in ROL.

Example 2.7 Following LDL rules are also ROL rules:

$$
\begin{array}{l}
ancestor(X, Y) \text{ :- } parent(X, Y) \\
ancestor(X, Y) \text{ :- } parent(X, Z), ancestor(Z, Y) \\
book\_deal(\{X, Y, Z\}) \text{ :- } book(X, Px), book(Y, Py), book(Z, Pz), \\
\qquad\qquad\qquad\qquad X \neq Y, X \neq Z, Y \neq Z, Px + Py + Pz < 100
\end{array}
$$

The complete set terms of ROL correspond to the set enumeration of LDL. The partial set terms of ROL is similar to the set grouping of LDL, however they cannot be used in constructed object terms for the reason discussed earlier. But the set grouping can be represented in ROL by using attributes. For example, the LDL rule $p(X, \langle Y \rangle)$:- $q(X, Y)$ can be represented in ROL using $p(X)[l \rightarrow \langle Y \rangle]$:- $q(X, Y)$. In ROL, we build in keys, single-valued and set-valued dependencies naturally.

As in [13, 37], we impose some restrictions on rules.

Definition 2.10 The *covered* terms of a rule are defined as follows:

(1) a value or an object identifier is covered;
(2) a term that is in a non-negative object expression in the body is covered;
(3) if $f(O_1, ..., O_n)$ is covered, then so is O_i, $1 \leq i \leq n$;
(4) if $O_1, ..., O_n$ are covered, then so are $f(O_1, ..., O_n)$, $\{O_1, ..., O_n\}$, and $\langle O_1, ..., O_n \rangle$;
(5) if δ_1 and δ_2 are covered, so is $\delta_1 o \delta_2$, $o \in \{+, -, \times, \div, \cup, \cap, \setminus\}$;
(6) if δ is covered, so is (δ);
(7) if $\delta_1 = \delta_2$ is in the body, then if δ_1 is covered, so is δ_2 (and vice versa);
(8) if $\delta_1 \in \delta_2$, $\delta_1 \subset \delta_2$, $\delta_1 \subseteq \delta_2$, $\delta_2 \supset \delta_1$, or $\delta_2 \supseteq \delta_1$ appears in the body, and δ_2 is covered, so is δ_1.

Definition 2.11 A rule is *safe* if all terms in the head are covered.

Following is an example of unsafe rule which defines an infinite set of pairs $loves(X, Y)$.

$loves(X, Y) : -X : person$

Definition 2.12 Let $K = (B, C, A, D_C, isa, D_A)$ be a schema. A *program* P based on K is a finite set of safe rules such that all class names and attribute labels in P are in $B \cup C \cup A$.

Syntactically, we just require that class names and attribute labels in P are in $B \cup C \cup A$. Semantically, the schema is used to constrain the objects and their attribute values generated from the program.

Definition 2.13 A *database* is a tuple $DB = (K, P)$, where K is a schema and P is a program based on K.

Definition 2.14 A *query* is a sequence of object or comparison expressions prefixed with ?–.

Example 2.8 The following example is a ROL database. It is taken from [9], where it is proposed as a task to test database languages.

$$
\begin{array}{ll}
\text{Schema} & part[name \Rightarrow string] \\
& quantity(part, integer) \\
& basepart[cost \Rightarrow integer(0..1000), mass \Rightarrow integer(0..10000)]\ isa\ part \\
& compositepart[madefrom \Rightarrow \{quantity(part, integer)\}, \\
& \qquad\qquad assemblycost \Rightarrow integer(0..5000), \\
& \qquad\qquad massincrement \Rightarrow integer(0..50000)]\ isa\ part \\
& \{quantity(part, integer)\}[totalcost \Rightarrow integer, totalmass \Rightarrow integer]
\end{array}
$$

Program
$$
\begin{aligned}
&p_1 : basepart[cost \rightarrow 20, mass \rightarrow 50]\\
&p_2 : basepart[cost \rightarrow 10, mass \rightarrow 30]\\
&p_3 : basepart[cost \rightarrow 15, mass \rightarrow 40]\\
&quantity(p_1, 1)\\
&quantity(p_2, 3)\\
&quantity(p_3, 2)\\
&quantity(p_4, 2)\\
&p_4 : compositepart[madefrom \rightarrow \{quantity(p_2, 3), quantity(p_3, 2)\}]\\
&p_5 : compositepart[madefrom \rightarrow \{quantity(p_1, 1), quantity(p_4, 2)\}]
\end{aligned}
$$

$$
\begin{aligned}
&\{quantity(P,\ Q)\}[totalcost \rightarrow C, totalmass \rightarrow M] \text{ :- } quantity(P,\ Q),\\
&\qquad P[cost \rightarrow C_1, mass \rightarrow M_1], C = Q \times C_1, M = Q \times M_1\\
&\{quantity(P,\ Q)\}[totalcost \rightarrow C, totalmass \rightarrow M] \text{ :- } quantity(P,\ Q),\\
&\qquad X[assemblycost \rightarrow C_1, massincrement \rightarrow M_1],\\
&\qquad C = Q \times C_1, M = Q \times M_1\\
&S[totalcost \rightarrow C, totalmass \rightarrow M] \text{ :- } S = S_1 \cup S_2, S_1 \cap S_2 = \{\},\\
&\qquad S_1[totalcost \rightarrow C_1, totalmass \rightarrow M_1],\\
&\qquad S_2[totalcost \rightarrow C_2, totalmass \rightarrow M_2],\\
&\qquad C = C_1 + C_2, M = M_1 + M_2\\
&P[assemblycost \rightarrow C, massincrement \rightarrow M] \text{ :- } P[madefrom \rightarrow S],\\
&\qquad S[totalcost \rightarrow C, totalmass \rightarrow M]
\end{aligned}
$$

This is a manufacturing company's parts database. There are two kinds of parts: base parts and composite parts. Both *basepart* and *compositepart* are subclasses of *part* and inherit the *name* attribute from *part*. In order to express the quantity of each part used to manufacture a composite part, a constructed object class $quantity(part, integer)$ is used. The set class $\{quantity(part, integer)\}$ has attributes *totalcost* and *totalmass*.

The facts in the program are about the base parts, their cost and mass, the quantity of parts used in manufacturing composite parts, and the way composite parts are made from other parts. The rules in the program tell how to compute the assembly cost and mass increment of composite parts.

To see how the assembly cost and mass increment of composite parts are derived using rules, let us consider the composite part p_4. Using the last rule and the fact that $p_4[madefrom \rightarrow \{quantity(p_2, 3), quantity(p_3, 2)\}]$, we just need to derive the *totalcost* of the set $\{quantity(p_2, 3), quantity(p_3, 2)\}$. With the second last rule, we can partition the set into $\{quantity(p_2, 3)\}$ and $\{quantity(p_3, 2)\}$ and directly derive their *totalcost* and *totalmass* with the first rule. We therefore derive p_4's cost 60 and mass 170. Similarly, we can derive p_5's cost 140 and mass 390.

Note that the following important integrity constraints specified in [9] are built-in in the schema of the above database without any additional efforts:

(1) Each base and composite part is also a part.
This is done directly by instance level inheritance.
(2) Composite parts and base parts are disjoint.
This is built-in into the semantics that an object can only have one immediate class.
(3) Only composite parts can be assembled.
This is direct from the schema.

The above example shows that the task of finding the assembly cost and mass increment of a composite part can be done in ROL not only declaratively, but also object-orientedly.

3 Semantics of ROL

In this section, we define the Herbrand interpretations and models for ROL programs. We first define the universe.

Definition 3.1 Let $K = (B, C, A, D_C, isa, D_A)$ be a schema. The *Herbrand universe* U_K based on K is defined as follows:

$U_0 = \mathcal{O} \cup \mathcal{D}$
$U_i = U_{i-1} \cup F(U_{i-1}) \cup S(U_{i-1})$
$U_K = \cup_{i=0}^{\infty} U_i$

where

$F(U) = \{f(o_1, ..., o_n) \mid f \in C \text{ is an n-ary object class name}, n \geq 1, o_i \in U, \text{ for } i = 1, ..., n\}$
$S(U) = \{\{o_1, ..., o_n\} \mid n \geq 0 \text{ and } o_i \in U, \text{ for } i = 1, ..., n\}$

Note that we can have nested sets but not infinite sets. By doing so, we avoid annoying problems when we try to compare interpretations. We will discuss this issue in the next section.

Definition 3.2 Let $K = (B, C, A, D_C, isa, D_A)$ be a schema. The *Herbrand base* B_K based on K is the set of all ground simple object expressions which can be formed by using class names and attribute labels in $B \cup C \cup A$ and objects in U_K.

Note that the Herbrand base does not contain ground composite object expressions or involve any partial sets. Therefore, it is not the set of all ground simple object expressions.

Definition 3.3 A subset of B_K is *consistent* if it does not contain a pair of ground object expressions $o[l \rightarrow o_1]$ and $o[l \rightarrow o_2]$ such that $o_1 \neq o_2$.

Definition 3.4 Let K be a schema. A *Herbrand interpretation* based on K is a consistent subset of B_K.

Before we define the notion of satisfaction of a rule, and thus of a program, we introduce several auxiliary notions. First, we extend the Herbrand universe U_K to include partial sets.

Definition 3.5 Let K be a schema and U_K a Herbrand universe based on K. The *extended Herbrand universe* U_K^+ based on K is defined as follows:

$$U_K^+ = U_K \cup \{\langle o_1, ..., o_n \rangle \mid \{o_1, ..., o_n\} \in U_K, n \geq 1\}.$$

Definition 3.6 Let $K = (B, C, A, D_C, isa, D_A)$ be a schema. The *extended Herbrand base* B_K^+ based on K is the set of all ground simple object expressions which can be formed by using class names and attribute labels in $B \cup C \cup A$ and objects in U_K^+.

The difference between B_K^+ and B_K is that partial sets can appear in object expressions in B_K^+ but not in B_K.

Definition 3.7 Let S be a subset of B_K^+. The *instances* of classes in S are defined as follows:

(1) A value is an instance of a value class in S iff it is an element in the collection which the value class denotes.
(2) An abstract object o is an instance of a class p in S iff $o : p \in S$.
(3) A constructed object $f(o_1, ..., o_n)$ is an instance of a class $f(p_1, ..., p_n)$ in S iff $f(o_1, ..., o_n) \in S$ and o_i is an instance of p_i in S for $i = 1, ..., n$.
(4) A set (complete or partial) s is an instance of a set class $\{p\}$ in S iff for every $o \in s$, o is an instance of p in S.

It is our intention to let every instance of a subclass be an instance of its superclasses. As a result, an object may be an instance of several classes.

Let o be an instance of the class p in S. Then o is an *immediate* instance of p in S (p is an *immediate* class of o in S) iff there does not exist a class q of which o is an instance and q *e-isa** p.

Example 3.1 Let $p, q, f(integer), g(integer), \{p\}, \{q\}, \{f(integer)\}, \{g(integer)\}$ be classes, assume that p *e-isa** $f(integer)$ and $g(integer)$ *e-isa** q and

$$S = \{o_1 : p, o_1 : f(integer), o_2 : f(integer), f(1), f(1) : g(integer), g(2), g(2) : q\}$$

Then o_1 is an instance of p and $f(integer)$ and is an immediate instance of p in S; o_2 is an immediate instance of $f(integer)$ in S; $f(1)$ is an immediate instance of $f(integer)$ and $g(integer)$ in S; $g(2)$ is an instance of $g(integer)$ and q in S and is an immediate instance of $g(integer)$ in S; $\langle g(2) \rangle$ is an instance of $\{g(integer)\}$ and $\{q\}$ and is an immediate instance of $\{g(integer)\}$; and $\{f(1), g(2)\}$ is an immediate instance of $\{q\}$.

Definition 3.8 Let $K = (B, C, A, D_C, isa, D_A)$ be a schema, S a subset of B_K^+, and ψ a ground simple object expression. Then ψ is *well-typed* with respect to K in S if one of the following holds:

(1) ψ is $o : p$ and o is an instance of p in S, o has a unique immediate class in S and if o is a constructed object or p is a constructed class, then o is not an immediate instance of p.
(2) ψ is $f(o_1, ..., o_n)$ and there exists a constructed object class $f(p_1, ..., p_n)$ in D_C such that $f(o_1, ..., o_n)$ is an instance of $f(p_1, ..., p_n)$ in S.
(3) ψ is $o_1[l \rightarrow o_2]$, there exists a unique class p such that o_1 is an immediate instance of p in S, there exists q such that $p[l \Rightarrow q] \in D_A^*$, and for all q such that $p[l \Rightarrow q] \in D_A^*$, o_2 is an instance of q.

Continue with the above example, $o_2 : f(integer)$ is not well-typed in S because o_2 is an immediate instance of $f(integer)$; $f(1) : g(integer)$ is not well-typed in S because $f(1)$ is also an immediate instance of $g(integer)$ besides $f(integer)$.

Definition 3.9 Let K be a schema and I an interpretation based on K. Then I is *well-typed* with respect to K iff every object expression of I is well-typed with respect to K in I.

Based on the above discussions, it is clear that in a well-typed interpretation, every object can have exactly one immediate class. In particular, an object identifier can only have an atomic object class as its immediate class and a constructed object $f(o_1, ..., o_n)$ can only have a constructed class of the form $f(p_1, ..., p_n)$ as its immediate class.

Definition 3.10 A *ground substitution* θ is a mapping from $\mathcal{V}$ to U_K. It is extended to terms, and expressions as follows:

(1) if $o \in \mathcal{O} \cup \mathcal{D}$ then $\theta o = o$
(2) if $X \in \mathcal{V}$ then $\theta X = \theta(X)$
(3) $\theta f(O_1, ..., O_n) = f(\theta O_1, ..., \theta O_n)$
(4) $\theta\{O_1, ..., O_n\} = \{\theta O_1, ..., \theta O_n\}$
(5) $\theta\langle O_1, ..., O_n\rangle = \langle \theta O_1, ..., \theta O_n\rangle$
(6) if E is an object expression or an arithmetic or set-theoretic comparison expression, then θE results from E by applying θ to every term in E.

Definition 3.11 Let K be a schema and I a well-typed interpretation with respect to K. The notion of satisfaction (denoted by $\models$) and its negation (denoted by $\not\models$) are defined as follows.

(1) Let o be an object and p a class. Then $I \models o : p$ iff o is an instance of p in I and for every q such that p *e-isa** q, o is also an instance of q in I.
(2) Let $f(o_1, ..., o_n)$ be a constructed object. Then $I \models f(o_1, ..., o_n)$ iff $f(o_1, ..., o_n) \in I$.
(3) For each ground simple object expression $o_1[l \rightarrow o_2]$
    - if o_2 is not a partial set, then $I \models o_1[l \rightarrow o_2]$ iff $o_1[l \rightarrow o_2] \in I$.
    - if o_2 is a partial set, then $I \models o_1[l \rightarrow o_2]$ iff there exists a complete set o_2' such that $I \models o_1[l \rightarrow o_2']$ and for each $o \in o_2$, $o \in o_2'$.
(4) For each ground composite object expression ψ, $I \models \psi$ iff for every constituent object expression φ of ψ, $I \models \varphi$.
(5) For each ground negative object expression $\psi = \neg\varphi$, $I \models \psi$ iff $I \not\models \varphi$.
(6) For each ground arithmetic and set-theoretic comparison expression ψ, $I \models \psi$ iff ψ is true in the standard arithmetic and set-theoretic interpretation.
(7) For each rule r of the form A :- $L_1, ..., L_n$, $I \models r$ iff for each ground substitution θ, $I \models \theta L_1$,..., $I \models \theta L_n$ implies $I \models \theta A$.
(8) For each program P, $I \models P$ iff for each rule $r \in P$, $I \models r$.

Note that although an interpretation does not contain expressions with partial sets, the satisfaction of expressions with partial sets is determined based on the expressions with complete sets in the interpretation. For example, if $tom[parents \rightarrow \{ann, max\}] \in I$, then $I \models tom[parents \rightarrow \langle ann\rangle]$, $I \models tom[parents \rightarrow \langle max\rangle]$, and $I \models tom[parents \rightarrow \langle ann, max\rangle]$.

Definition 3.12 A *model* M of a program P is a well-typed interpretation which satisfies P.

Because of the typing and consistency constraints, not all ROL programs have models.

Example 3.2 Consider the following database:

Schema	$person[mother \Rightarrow person, age \Rightarrow integer(0..125)]$
Program	$tom : person[mother \rightarrow tom]$
	$tom : person[mother \rightarrow ann]$
	$ann : person[age \rightarrow 126]$

In the program, the object expressions $tom[mother \rightarrow tom]$ and $tom[mother \rightarrow ann]$ violate the consistency constraint, while the object expression $ann[age \rightarrow 126]$ violates the typing constraint. Therefore, no well-typed interpretation can satisfy this program and thus the database.

However, a database may have several different interpretations and models.

Example 3.3 The following database will be used as a running example for the rest of the paper:

Schema	$person[parents \Rightarrow \{person\}]$
	$family(\{person\})[children \Rightarrow \{person\}]$
Program	$ann : person$
	$tom : person[parents \rightarrow \langle ann \rangle]$
	$family(S)[children \rightarrow \langle X \rangle] \text{ :- } X[parents \rightarrow S]$

The program says ann and tom are $person$, ann is one of tom's parents, and if X has parents S, then the family denoted by the constructed object $family(S)$ has X as one of its children.

A well-typed interpretation I for the program is

$$I = \{ann : person, tom : person, tom[parents \rightarrow \{ann\}], \\ family(\{ann\}), family(\{ann\})[children \rightarrow \{tom\}]\}$$

Indeed, the interpretation I is a model of the program by the definition. The following well-typed interpretations I_1 and I_2 are also models:

$$I_1 = \{ann : person, tom : person, max : person, tom[parents \rightarrow \{ann, max\}], \\ family(\{ann, max\}), family(\{ann, max\})[children \rightarrow \{tom\}]\}$$

$$I_2 = \{ann : person, tom : person, sam : person, tom[parents \rightarrow \{ann, tom\}], \\ family(\{ann, tom\}), family(\{ann, tom\})[children \rightarrow \{tom, sam\}]\}$$

Because a ROL database may have several different interpretations and models, which means that it can be interpreted differently, what is the semantics of the program then? Intuitively, I is "smaller" than I_1 and I_2, but it is not a subset of either I_1 or I_2.

In Datalog, a well-defined and well-justified minimal model is selected as the intended semantics of the program as it is the natural consequence of the closed world assumption. In ROL, we will also use a well-defined and well-justified minimal model as the intended semantics of the program. However, with sets in interpretations, the notion of minimal model is not well-defined as we don't have a proper order on interpretations and models.

4 Comparing Interpretations

In this section, we study how different objects, object expressions, interpretations, and models can be compared so that we can properly define the minimal model of a program. As interpretations only involve compact objects, we use objects to refer to compact objects in this section for convenience.

Definition 4.1 Let K be a schema. An object $o \in U_K$ is a *sub-object* of an object $o' \in U_K$, denoted by $o \preceq o'$, iff:

(1) both are elements of U_0, and they are equal;
(2) $o = f(o_1, ..., o_n)$, $o' = f(o'_1, ..., o'_n)$, such that $o_i \preceq o'_i$ for $i = 1, ..., n$.
(3) both are complete sets and for all $o_i \in o$, there exists $o'_i \in o'$, such that $o_i \preceq o'_i$;

For example, $ann \preceq ann$, $\{ann\} \preceq \{ann, max\}$, $family(\{ann\}) \preceq family(\{ann, max\})$.

This sub-object relationship is similar to the sub-object relationship of Bancilhon-Koshafian [11].

We first introduce the notion of the depth of an object as in [11], which we will use in some of our proofs in this section.

Definition 4.2 The depth of an object o is defined as follows:

(1) if o is an element of U_0, then $depth(o) = 1$;
(2) if $o = f(o_1, ..., o_n)$, then $depth(o) = max\{depth(o_i) \mid i = 1, ..., n\} + 1$.
(3) if $o = \{o_1, ..., o_n\}$, then $depth(o) = max\{depth(o_i) \mid i = 1, ..., n\} + 1$ if $n > 0$; otherwise, $depth(o) = 2$.

The sub-object relationship has the following property as in [11].

Proposition 4.1 The sub-object relationship is reflexive and transitive.

Proof. Straightforward by induction on the depth of objects. □

However, anti-symmetry does not hold. For example, let $o = \{family(\{ann\})\}$ and $o' = \{family(\{ann\}), family(\{\})\}$. Then $o \preceq o'$ and $o' \preceq o$ but $o \neq o'$. The problem is that o' has redundant information $family(\{\})$ which is a sub-object of $family(\{ann\})$ in o'. The redundant information may be in some deeply nested complete sets or constructed objects.

In [11], the notion of *reduced objects* is introduced to solve this problem. A set is reduced iff it doesn't contain two distinct objects o_1 and o_2 such that $o_1 \preceq o_2$. For example, $o' = \{family(\{ann\}), family(\{\})\}$ is not reduced. By requiring all objects to be reduced, we obtain a partial order on objects which is referred to as the Hoare ordering in the literature. However, such a notion is too restrictive as it disallows meaningful objects like $\{\{english\}, \{english, french\}\}$ to exist in the database.

In [15] and [36], the notion of *cochain* is used. Two objects o_1 and o_2 are considered equivalent if $o_1 \preceq o_2$ and $o_2 \preceq o_1$. Let $[o]$ denote an equivalence class. The sub-object relationship is then extended to equivalence classes by $[o_1] \preceq [o_2]$ iff $o_1 \preceq o_2$. The set of equivalence classes is called a cochain. Thus on a cochain the sub-object relationship is a partial order. As a result, non-reduced objects can exist in the database and their semantics is given by their equivalent classes. For example, $\{\{english\}, \{english, french\}\}$ can exist in the database and has the same semantics as $\{\{english, french\}\}$. This approach still cannot capture the intended semantics of objects that involve nested sets.

Example 4.1 Consider the following database:

Schema	$language[\,]$
	$person[speaks \Rightarrow \{language\}]$
	$survey[l \Rightarrow \{\{language\}\}]$
Program	$english : language$
	$french : language$
	$ann : person[speaks \rightarrow \{french\}]$
	$tom : person[speaks \rightarrow \{english, french\}]$
	$s1 : survey[l \rightarrow \langle Languages \rangle] :\!- Person[speaks \rightarrow Languages]$

Using the rule in the program, we can obtain the set of sets of languages that each person speaks: $\{\{english\}, \{english, french\}\}$. Although $\{english\} \preceq \{english, french\}\}$ in this set, it is not redundant at all. The set $\{\{english\}, \{english, french\}\}$ here has a meaning quite different from $\{\{english, french\}\}$ and $\{\{english\}, \{french\}, \{english, french\}\}$. It should be interpreted by itself rather than having the same semantics as $\{\{english, french\}\}$ and $\{\{english\}, \{french\}, \{english, french\}\}$.

We now introduce a new powerful notion that can capture the intended semantics of objects involving nested sets.

Definition 4.3 Let K be a schema. An object $o \in U_K$ is a *preferable* sub-object of another object $o' \in U_K$, denoted by $o \preceq_p o'$, iff:

(1) both are elements of U_0, and they are equal;
(2) $o = f(o_1, ..., o_n)$, $o' = f(o'_1, ..., o'_n)$, such that $o_i \preceq_p o'_i$ for $i = 1, ..., n$.
(3) both are complete sets, and for each $o_i \in o - o'$, there exists $o'_i \in o' - o$, such that $o_i \preceq_p o'_i$;

For example, we have

$\{family(\{ann\})\} \preceq_p \{family(\{ann\}), family(\{\})\}$
$\{family(\{ann\}, family(\{\})\} \not\preceq_p \{family(\{ann\})$
$\{\{english\}, \{english, french\}\} \preceq_p \{\{english\}, \{french\}, \{english, french\}\}$
$\{\{english\}, \{french\}, \{english, french\}\} \not\preceq_p \{\{english\}, \{english, french\}\}$

Proposition 4.2 The preferable sub-object relationship over is reflexive and transitive.

Proof. Reflexivity is obvious. We prove transitivity by induction on the depth of objects.

For objects of depth 1, transitivity clearly holds.

Assume that transitivity holds for objects whose depth is less than or equal to m with $m \geq 1$. We now prove that transitivity holds for objects of depth equal to $m + 1$. Let o, o' and o'' be three objects of depth $m + 1$ such that $o \preceq_p o'$ and $o' \preceq_p o''$. Then they are either all complete sets or all constructed objects.

Assume that they are all complete sets. Let $o_1 \in o - o''$. Two situations can occur: $o_1 \notin o'$ and $o_1 \in o'$.

Assume first $o_1 \notin o'$, we have $o_1 \in o - o'$. Then there exists $o_2 \in o' - o$ such that $o_1 \preceq_p o_2$. If $o_2 \in o''$, then $o_2 \in o'' - o$ which is what we want. If $o_2 \notin o''$, then $o_2 \in o' - o''$ and there exists $o_3 \in o'' - o'$ such that $o_1 \preceq_p o_2 \preceq_p o_3$. If $o_3 \notin o$, then $o_3 \in o'' - o$ and we are done. Assume

$o_3 \in o$, then $o_3 \in o - o'$. By repeating the above process, we can obtain a sequence of distinct objects $o_1, o_2, o_3, o_4, ...$ such that $o_1 \preceq_p o_2 \preceq_p o_3 \preceq_p o_4 ...$. Since o, o' and o'' are finite sets, the sequence must be finite as well. Therefore, there exists an o_n with $n \geq 2$ such that $o_n \in o'' - o$ and $o_1 \preceq_p o_2 \preceq_p o_3 ... \preceq_p o_n$. This is true for the case $o_1 \in o'$ as well. By the induction hypothesis, we have $o_1 \preceq_p o_n$. This yields $o \preceq_p o''$.

Now assume that they are all constructed objects. Let $o = f(o_1, ..., o_n)$, $o' = f(o'_1, ..., o'_n)$, and $o'' = f(o''_1, ..., o''_n)$. Since $o \preceq_p o'$ and $o' \preceq_p o''$, we have $o_i \preceq_p o'_i$ and $o'_i \preceq_p o''_i$ for $i = 1, ..., n$. By the induction hypothesis, we have $o_i \preceq_p o''_i$ for $i = 1, ..., n$. Therefore $o \preceq_p o''$.□

Proposition 4.3 The preferable sub-object relationship is anti-symmetric.

Proof. The proof is again by induction on the depth of objects.

The property clearly holds for compact objects of depth 1.

Assume that anti-symmetry holds for compact objects whose depth is less than or equal to m with $m \geq 1$. We prove now that the property holds for compact objects of depth equal to $m + 1$. Let o and o' be two compact objects of depth $m + 1$ such that $o \preceq_p o'$ and $o' \preceq_p o$. Two situations can occur: they are either both complete sets or both constructed objects.

Assume first that they are both complete sets. Let $o_1 \in o - o'$. Then there exists $o_2 \in o' - o$ such that $o_1 \preceq_p o_2$. Since $o' \preceq_p o$ and $o_2 \in o' - o$, there exists $o_3 \in o - o'$ such that $o_2 \preceq_p o_3$. Therefore we can find two infinite and disjoint sequence of objects $o_1, o_3, ...$ in $o - o'$ and $o_2, o_4, ...$ in $o' - o$ such that $o_1 \preceq_p o_2 \preceq_p o_3 \preceq_p o_4 ...$. Now we prove $o_1, o_2, ...$, are all distinct. Suppose that $o_i \preceq_p o_j \preceq_p ... \preceq_p o_i$. Since $\preceq_p$ is transitive, we have $o_i \preceq_p o_j$ and $o_j \preceq_p o_i$. By the induction hypothesis, we have $o_i = o_j$, which is a contradiction. Thus, o and o' must both be infinite sets of objects, which is again a contradiction. Therefore, there exists no element in $o - o'$. Similarly, we can prove there exists no element in $o' - o$. This yields $o = o'$.

Now assume that they are both constructed objects and let $o = f(o_1, ..., o_n)$ and $o' = f(o'_1, ..., o'_n)$. Since $o \preceq_p o'$ and $o' \preceq_p o$, we have $o_i \preceq_p o'_i$ and $o'_i \preceq_p o_i$ for $i = 1, ..., n$. By the induction hypothesis, we have $o_i = o'_i$ for $i = 1, ..., n$. Therefore $o = o'$.□

Therefore, from Propositions 4.2 and 4.3, we have

Proposition 4.4 The preferable sub-object relationship is a partial order.

Based on the preferable sub-object relationship, we now define the sub-object expression relationship.

Definition 4.4 The *preferable sub-object expression* relationship between object expressions of B_K, denoted by $\preceq_p$, is defined as follows.

(1) $o_1 : p \preceq_p o_2 : p$, if $o_1 \preceq_p o_2$.
(2) $f(o_1, ..., o_n) \preceq f(o'_1, ..., o'_n)$, if $o_i \preceq_p o'_i$ for $i = 1, ..., n$.
(3) $o_1[l \rightarrow o_2] \preceq_p o'_1[l \rightarrow o'_2]$, if $o_1 \preceq_p o'_1$ and $o_2 \preceq_p o'_2$.

For example, we have

$ann : person \preceq_p ann : person$
$family(\{ann\}) \preceq_p family(\{ann, sam\})$
$g(family(\{ann\}), 1) : p \preceq_p g(family(\{ann, sam\}), 1) : p$

$family(\{ann\})[children \to \{tom\}] \preceq_p family(\{ann, tom\})[children \to \{tom, sam\}]$
$s_1[l \to \{\{english\}, \{english, french\}\}] \preceq_p s_1[l \to \{\{english\}, \{french\}, \{english, french\}\}$

The sub-object expression relationship has the following property.

Proposition 4.5 The preferable sub-object expression relationship is a partial order.

Proof. Straightforward from Definition 4.4 and Proposition 4.4. □

Definition 4.5 Let I_1 and I_2 be two interpretations. Then I_1 is a *sub-interpretation* of I_2, or I_2 *contains* I_1, denoted by $I_1 \sqsubseteq I_2$, iff for every object expression $\psi_1 \in I_1$, there exists an object expression $\psi_2 \in I_2$ such that $\psi_1 \preceq_p \psi_2$.

For the interpretations I, I_1, I_2 in Example 3.3, we have $I \sqsubseteq I_1$, $I \sqsubseteq I_2$ but $I_1 \not\sqsubseteq I_2$, $I_2 \not\sqsubseteq I_1$.

Proposition 4.6 The sub-interpretation relationship is reflexive and transitive.

Proof. Straightforward from Definition 4.5 and Proposition 4.5. □

For a similar reason, anti-symmetry does not hold here either. For example, let $I_1 = \{p(\{1,2\},5)\}$ and $I_2 = \{p(\{1,2\},5), p(\{1\},5)\}$, we have $I_1 \sqsubseteq I_2 \sqsubseteq I_1$ but $I_1 \neq I_2$.

As for the sub-object relationship, we introduce the notion of preferable sub-interpretation.

Definition 4.6 Let I_1 and I_2 be interpretations. Then I_1 is a *preferable* sub-interpretation to I_2, denoted by $I_1 \sqsubseteq_p I_2$, iff for every object expression $\psi_1 \in I_1 - I_2$, there exists an object expression $\psi_2 \in I_2 - I_1$ such that $\psi_1 \preceq_p \psi_2$.

Proposition 4.7 Let L be the set of all interpretations. Then $\sqsubseteq_p$ is a partial order on L.

Proof. a) Reflexivity is obvious.

b) Transitivity. Let I_1, I_2 and I_3 be interpretations such that $I_1 \sqsubseteq_p I_2$ and $I_2 \sqsubseteq_p I_3$. We prove $I_1 \sqsubseteq_p I_3$. Let $\psi_1 \in I_1 - I_3$. Two situations can occur: $\psi_1 \notin I_2$ and $\psi_1 \in I_2$. Assume first $\psi_1 \notin I_2$, we have $\psi_1 \in I_1 - I_2$. Then there exists $\psi_2 \in I_2 - I_1$ such that $\psi_1 \preceq_p \psi_2$. If $\psi_2 \in I_3$, then $\psi_2 \in I_3 - I_1$ which is what we want. Assume $\psi_2 \notin I_3$, then $\psi_2 \in I_2 - I_3$ and there exists $\psi_3 \in I_3 - I_2$ such that $\psi_1 \preceq_p \psi_2 \preceq_p \psi_3$. If $\psi_3 \notin I_1$, then $\psi_3 \in I_3 - I_1$ and we are done. Assume $\psi_3 \in I_1$, then $\psi_3 \in I_1 - I_2$. By repeating the above process, we can obtain a sequence of distinct object expressions $\psi_1, \psi_2, \psi_3, ...$ such that $\psi_1 \preceq_p \psi_2 \preceq_p \psi_3 ...$. We now prove that the sequence must be finite even though I_1, I_2 and I_3 can be infinite. Assume it is not finite. Since ψ_i can be one of the forms $o_i : p$, $f(o_1, ..., o_n)$, and $o_i[l \to o_i']$, we assume first ψ_i is of the form $o_i : p$. Then we have an infinite sequence of distinct objects $o_1, o_2, ...$ such that $o_1 \preceq_p o_2 \preceq_p o_3 \preceq_p o_4 ....$ If all o_i are elements of U_0, then it is impossible to have such a sequence. If all o_i are sets or constructed objects, then this infinite sequence implies we have infinite sets, which is a contradiction. The cases in which all ψ_i are of the form $f(o_1, ..., o_n)$ and $o_i[l \to o_i']$ are similar. Therefore, there exists a ψ_n with $n \geq 2$ such that $\psi_n \in I_3 - I_1$ and $\psi_1 \preceq_p \psi_2 \preceq_p \psi_3 ... \preceq_p \psi_n$. This is also true for the case $\psi_1 \in I_2$. By Proposition 4.5, we have $\psi_1 \preceq_p \psi_n$. This yields $I_1 \sqsubseteq_p I_3$.

c) Anti-symmetry. The proof is straightforward based on the proofs for transitivity above and for Proposition 4.3. □

Therefore by requiring an interpretation to be preferable to other interpretations, we factor out problems such as the one presented above.

Definition 4.7 A model M of P is *minimal* iff for each model N of P, if $N \sqsubseteq_p M$ then $N = M$.

For the interpretations I, I_1, I_2 in Example 3.3, I is a minimal model. For the program in Example 4.1, its minimal model contains

$$s_1[l \rightarrow \{\{english\}, \{english, french\}\}]$$

Therefore, the semantics of the set $\{\{english\}, \{english, french\}\}$ is given by itself in this minimal model.

The notion of preferable sub-interpretation is similar to the notion of d-preferability ($\leq_{d,M}$) in LDL [13]. However, the model minimality in LDL is ill-defined based on $\leq_{d,M}$. First, $\leq_{d,M}$ is irreflexive by definition. Besides, $\leq_{d,M}$ is not transitive since nested infinite sets are allowed. Finally, $\leq_{d,M}$ is not anti-symmetric since there isn't an equivalent notion of preferable sub-objects. For example, let $M_1 = \{p(\{\{1\}, \{1, 2\}\})\}$ and $M_2 = \{p(\{\{1, 2\}\})\}$. Then $M_1 \leq_{d,M} M_2 \leq_{d,M} M_1$, but $M_1 \neq M_2$.

Definition 4.8 Let $DB = (K, P)$ be a database and I_1 and I_2 be two well-typed interpretations of P with respect to schema K. The *intersection* I, of I_1 and I_2, denoted by $I = I_1 \sqcap I_2$, is defined as follows:

(1) if $o : p \in I_1$ and $o : p \in I_2$, then $o : p \in I$,
(2) if $f(o_1, ..., o_n) \in I_1$ and $f(o_1, ..., o_n) \in I_2$, then $f(o_1, ..., o_n) \in I$,
(3) if $o_1[l_f \rightarrow o_2] \in I_1$ and $o_1[l_f \rightarrow o_2] \in I_2$, where l_f is a single-valued attribute, then $o_1[l_f \rightarrow o_2] \in I$,
(4) if $o[l_s \rightarrow o_1] \in I_1$ and $o[l_s \rightarrow o_2] \in I_2$, where l_s is a set-valued attribute and o_1 and o_2 are both complete sets, then $o[l_s \rightarrow o_3] \in I$ where $o_3 = o_1 \cap o_2$.

The intersection of interpretations of a program has the following properties.

Proposition 4.8 Let $DB = (K, P)$ be a database and I the intersection of two interpretations I_1 and I_2 of P. Then

(1) I is an interpretation.
(2) $I \sqsubseteq_p I_1$ and $I \sqsubseteq_p I_2$.
(3) if I_1 and I_2 are both well-typed interpretations with respect to schema K, then I is also a well-typed interpretation with respect to K.

Proof. Direct from Definitions 3.4, 3.9, and 4.8. □

Proposition 4.9 The relation $\sqcap$ over interpretations of a given program is commutative, idempotent and associative, i.e., $I_1 \sqcap I_2 = I_2 \sqcap I_1$, $I_1 \sqcap I_1 = I_1$, and $I_1 \sqcap (I_2 \sqcap I_3) = (I_1 \sqcap I_2) \sqcap I_3$ for any interpretations I_1, I_2 and I_3 of some program.

Proof. Direct from Definition 4.8. □

The intersection of two models is not necessary a model. For example, the intersection of two models I_1 and I_2 in Example 3.3 is $\{ann : person, tom : person, tom[parents \rightarrow \{ann\}]\}$, which is not a model.

The notion of intersection is used in the next section.

5 Bottom-Up Semantics

In this section, we show that under a suitable syntactic restriction on the program, namely, stratification, a well-defined and well-justified minimal model, when it exists, can be computed using a sequence of fixpoint operators and used as the intended semantics of the program.

5.1 Stratification

The notion of stratification has been used to give semantics to programs involving negation and sets [2, 8, 13]. We present a similar notion here which takes partial set terms into account.

Let $K = (B, C, A, D_C, isa, D_A)$ be a schema and P a program based on K. The set D_P of *defined symbols* of P is defined as $D_P = B \cup C \cup A$. The *defined symbols* of a rule is the defined symbols of the head of the rule.

Unlike Datalog, Prolog, LDL, COL in which a rule has only one defined symbol, a ROL rule may have several defined symbols. For example, the rule

$$jack : person[age \rightarrow X, parents \rightarrow S] \text{ :- } john[age \rightarrow X, parents \rightarrow S]$$

has defined symbols $person, age, parents$.

Definition 5.1 Let P be a program based on a schema K, and D_P the set of defined symbols of P. The relationships $>$, and $\geq$ on D_P are defined as follows:

(1) $x > y$ if there is a rule in which x is in the head, y in the body, and one of the following holds:

- y is in a negative object expression;
- y is in an object expression $P[y \rightarrow Q]$ such that Q is a complete set term and there exists a rule such that $P'[y \rightarrow \langle Q_1, ..., Q_n \rangle]$ is a constituent object expression of the head.

(2) $x \geq y$ if there is a rule in which x is in the head, y is in the body, and $x > y$ is not true.

Let $P[l \rightarrow Q]$ be in the body of a rule r_1, and Q a complete set term. If there exists a rule r_2 in which $P'[l \rightarrow \langle Q_1, ..., Q_n \rangle]$ is in the head, then if we use the rule r_1 before or at the same time with the rule r_2, the value for the set-valued attribute l may not be finalized. Therefore, the defined symbols of the head of r_1 must be at a higher level than l so that r_2 must be used before r_1. The case for negation is the same.

Consider the program in Example 3.3, $D_P = \{person, parents, family, children\}$ and $family > parents$, $children > parents$.

Definition 5.2 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 x to y if $x \geq y$, and
(3) there is a marked edge from x to y if $x > y$.

A dependency graph of P represents the relationship *depends-on* between defined symbols of P.

Definition 5.3 A program P is *stratified* iff its dependency graph G_P has no cycle with a marked edge.

Consider the following four rules:

$$X[ancestors \rightarrow \langle Y \rangle] \text{ :- } X[parents \rightarrow \langle Z \rangle], Z[ancestors \rightarrow \langle Y \rangle]$$
$$X[ancestors \rightarrow \langle Y \rangle] \text{ :- } X[parents \rightarrow S], Z \in S, Z[ancestors \rightarrow \langle Y \rangle].$$
$$X[ancestors \rightarrow \langle Y \rangle] \text{ :- } X[parents \rightarrow \langle Z \rangle], Z[ancestors \rightarrow S], Y \in S.$$
$$X[ancestors \rightarrow \langle Y \rangle] \text{ :- } X[parents \rightarrow S_1], Z \in S_1, Z[ancestors \rightarrow S_2], Y \in S_2.$$

Semantically, they all are equivalent. However, the first two rules are stratified, while the rest are not. Therefore, partial set terms in the body of rules cannot simply be replaced by set-valued variables and the set membership operation.

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

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

Proposition 5.1 Let P be a program, and D_P be the set of all defined symbols of P. Then P is stratified if and only if there is a partition $D_P = D_1 \dot{\cup} ... \dot{\cup} D_n$ such that
(1) if $x \in D_i$ and $x > y$, then there exists a j such that $i > j$ and $y \in D_j$.
(2) if $x \in D_i$ and $x \geq y$, then there exists a j such that $i \geq j$ and $y \in D_j$.

Each partition of symbols induces a partition of the program into strata. For each $D_P = D_1 \dot{\cup} ... \dot{\cup} D_n$, let $P = P_1 \dot{\cup} ... \dot{\cup} P_n$, where for each i,

$$P_i = \{r \in P \mid \text{the defined symbols of } r \text{ is in } D_i\}$$

Consider the program given in Example 3.3, it has two strata:

$$P_1 = \{ann : person, tom : person[parents \rightarrow \langle ann \rangle]\}$$
$$P_2 = \{family(S)[children \rightarrow \langle X \rangle] \text{ :- } X[parents \rightarrow S]$$

However, if the rule $tom : person[parents \rightarrow \langle ann \rangle]$ is replaced by $tom : person[parents \rightarrow \{ann\}]$, then the program just has one stratum.

5.2 Bottom-Up Semantics of Stratified Programs

In this section, we discuss how to define the semantics of stratified programs.

Definition 5.4 Let $DB = (K, P)$ be a database and I a well-typed interpretation of P w.r.t. K, the operator T_P of P over I is defined as follows.

$$T_P^1(I) = \{\theta B \mid A \text{ :- } L_1, ..., L_n \in R, B \text{ is a constituent expression of } A, \text{ and there exists a ground substitution } \theta \text{ such that } I \models \theta L_1, ..., I \models \theta L_n\}$$
$$T_P^2(I) = \{o : q \mid o : p \in T_P^1(I) \text{ and } p \; isa^* \; q\}$$
$$T_P(I) = T_P^1(I) \cup T_P^2(I)$$

if $T_P(I) \cup I$ is well-typed with respect to K; otherwise T_P is undefined.

Here T_P^1 is similar to the traditional immediate consequence operator, while T_P^2 performs instance level inheritance. That is, every instance of a subclass is also an instance of its superclasses. For example, if $mary : student \in T_P^1(I)$ and $student\ isa\ person$, then, $mary : person \in T_P^2(I)$. Note that our notion of T_P incorporates typing constraints and $T_P(I)$ may contain object expressions that are not compact.

Example 5.1 Consider the following database.

Schema	$person[parents \Rightarrow \{person\}]$
Program	$joe[parents \rightarrow \langle jim \rangle]$
	$joe[parents \rightarrow \langle X \rangle] \text{ :- } tom[parents \rightarrow \langle X \rangle]$

Let $I = \{ann : person, jim : person, joe : person, tom : person, tom[parents \rightarrow \{ann\}]\}$. Then I is a well-typed interpretation of the program. We have $T_P(I) = \{joe[parents \rightarrow \langle jim \rangle], joe[parents \rightarrow \langle ann \rangle]\}$, which is not an interpretation because it contains object expressions involving partial sets.

We will introduce a new operator called compaction operator which can be used to convert $T_P(I)$ into an interpretation. First, we introduce following auxiliary notions.

Definition 5.5 An object o is *part-of* a compact object o', denoted by $o \lhd o'$, iff one of the following holds:

(1) $o = o'$ where o is an object identifier, a constructed object or a complete set;
(2) o is a partial set and o' is a complete set, and for each $o_i \in o$, $o_i \in o'$.

Definition 5.6 An object expression ψ is *part-of* an object expression ψ', denoted by $\psi \lhd \psi'$, iff one of the following holds:

(1) $\psi = \psi'$ where ψ is either $o : p$ or $f(o_1, ..., o_n)$;
(2) $\psi = o_1[l \rightarrow o_2]$ and $\psi' = o_1[l \rightarrow o_2']$ and $o_2 \lhd o_2'$.

A set S of ground simple object expressions is *part-of* a ground simple object expression ψ, denoted by $S \lhd \psi$, iff for each $\psi' \in S$, $\psi' \lhd \psi$.

Following are several examples:

$\{jim : person\} \lhd jim : person$
$\{f(1) : g(integer)\} \lhd f(1) : g(integer)$
$\{family(\{ann\})\} \lhd family(\{ann\})$
$\{joe[parents \rightarrow \langle jim \rangle], joe[parents \rightarrow \langle ann \rangle]\} \lhd joe[parents \rightarrow \{ann, jim\}]$

Definition 5.7 The *compaction* operator C is a partial mapping from B_K^+ to B_K as follows (where S is any set of ground simple expressions).

$$
\begin{aligned}
C(S) = \ & \{o : p \in S\} \cup \\
& \{o[l \rightarrow o'] \in S \mid o' \text{ is a complete set}\} \cup \\
& \{o[l \rightarrow o'] \mid o' = \cup\{\{o_1, ..., o_n\} \mid o[l \rightarrow \langle o_1, ..., o_n \rangle] \in S\} \text{ and there exists} \\
& \qquad \text{no complete set } o'' \text{ such that } o' \subseteq o'' \text{ and } o[l \rightarrow o''] \in S\}
\end{aligned}
$$

if such obtained set is consistent; otherwise C is undefined.

Consider the following sets:

$S_1 = \{joe[parents \to \langle jim \rangle], joe[parents \to \langle ann \rangle]\}$
$S_2 = \{joe[parents \to \langle jim \rangle], joe[parents \to \{jim, ann\}]\}$
$S_3 = \{joe[parents \to \langle jim \rangle], joe[parents \to \langle ann \rangle], joe[parents \to \{jim, mary\}]\}$

Then $C(S_1) = C(S_2) = \{joe[parents \to \{jim, ann\}]\}$, and $C(S_3)$ is undefined.

Continue with Example 5.1, $C(T_P(I)) = \{joe[parents \to \{jim, ann\}]\}$ which is an interpretation of the program.

As above example shows, partial set terms in ROL 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 compaction operator C. 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 incorporated into the notion of satisfaction.

Proposition 5.2 The operator C has the following properties.

(1) if I is an interpretation, then $C(I) = I$;
(2) if $I \subseteq J$ and C is defined on both I and J, then $C(I) \sqsubseteq_p C(J)$.
(3) if C is defined on $I \cup J$ and the defined symbols of I and J are disjoint, then $C(I \cup J) = C(I) \cup C(J)$.

Proof. (1) Direct from Definition 5.7.

(2) For each $\psi \in C(I) - C(J)$ there exists a subset S of I such that $S \triangleleft \psi$ and ψ is the result of applying C to S. Since $I \subseteq J$, there exists a subset S' of J such that $S \subseteq S'$ and there exists a $\psi' \in C(J) - C(I)$, such that $S' \triangleleft \psi'$, and $\psi \preceq \psi'$. Therefore, $C(I) \sqsubseteq_p C(J)$.

(3) Direct from Definition 5.7. □

The operators C and T_P together have the following property.

Proposition 5.3 Let $DB = (K, P)$ be a database and M a well-typed interpretation with respect to K. If M is a model of P, then $C(T_P(M)) \sqsubseteq_p M$.

Proof. Let $\psi \in C(T_P(M))$. (1) If $\psi \in T_P^1(M)$, then there exists a rule $A :\!- L_1, ..., L_n$ and a ground substitution θ such that $M \models \theta L_1, ..., M \models \theta L_n$ and ψ is a ground constituent expression of θA. Since M is a model of P, $M \models \psi$. By Definition 3.11, $\psi \in M$. (2) If $\psi \in T_P^2(M)$, then there exists a corresponding $\psi' \in T_P^1(M)$. By (1) and Definition 3.11, $\psi \in M$. (3) If $\psi \notin T_P(M)$, then ψ is the form $o[l_s \to o']$ where o' is a complete set and there exists a subset S of $T_P(M)$ such that $S \triangleleft \psi$ and ψ is the result of applying C to S. For each $o[l_s \to \langle o_1, ..., o_n \rangle] \in S$, there exists a rule $A :\!- L_1, ..., L_n$ and a ground substitution θ such that $M \models \theta L_1, ..., M \models \theta L_n$ and $o[l_s \to \langle o_1, ..., o_n \rangle]$ is a ground constituent expression of θA. Since M is a model of P, $M \models o[l_s \to \langle o_1, ..., o_n \rangle]$. Then there exists a o'' such that $\{o_1, ..., o_n\} \subseteq o''$ and $o[l_s \to o''] \in M$. So $o' \subseteq o''$. Let ψ' be $o[l_s \to o'']$. Then $\psi \preceq \psi'$. Therefore, $C(T_P(M)) \sqsubseteq_p M$. □

Note that C and T_P together function as the T_P operator of a logic program [30]. For a logic program, if $T_P(M) \subseteq M$ then M is a model. Because of the existence of sets, such property no longer holds for ROL programs. That is, if $C(T_P(M)) \sqsubseteq_p M$, M is not necessarily a model.

Example 5.2 Consider the following database:

Schema $person[parents \Rightarrow \{person\}]$
Program $joe : person[parents \rightarrow \{mary\}]$

Let $M = \{joe : person, mary : person, joe[parents \rightarrow \{joe, mary\}]\}$ be a well-typed interpretation. Then $C(T_P(M)) = \{joe : person, joe[parents \rightarrow \{mary\}] \sqsubseteq_p M$, but M is obviously not a model of the program.

In a logic program with negation, there exist several minimal models. If every ground fact in a minimal model can be inferred from the program, then the minimal model is called supported and is selected as the semantics of the logic program [8]. We follow the classical approach of logic programming to defining the semantics of ROL program.

Definition 5.8 Let $DB = (K, P)$ be a database. A well-typed interpretation I w.r.t. K is *supported* iff $I \sqsubseteq_p C(T_P(I))$.

Proposition 5.4 Let $DB = (K, P)$ be a database and M a model of P. Then M is supported iff $M = C(T_P(M))$.

Proof. Direct from Definition 5.8 and Proposition 5.3.

Unlike traditional logic programs, a ROL database may have a unique minimal model which is not supported, because of the typing constraint.

Example 5.3 Consider the following database:

Schema $person[parents \Rightarrow \{person\}]$
Program $joe : person[parents \rightarrow \{mary\}]$

This program has a unique minimal model $\{joe : person, mary : person, joe[parents \rightarrow \{mary\}]\}$ but not supported since we cannot infer $mary : person$ from the program.

Definition 5.9 The powers of the operator T_P are defined using the compaction operator as follows:

$T_P \uparrow 0(I) = I$
$T_P \uparrow n(I) = T_P(C(T_P \uparrow (n-1)(I))) \cup T_P \uparrow (n-1)(I)$ if C is defined on $T_P \uparrow (n-1)(I)$
$T_P \uparrow \omega(I) = \cup_{n=0}^{\infty} T_P \uparrow n(I)$

If C is not defined on $T_P \uparrow (n-1)(I)$, then $T_P \uparrow (n+1)(I)$ is undefined.

Note that T_P is defined only on interpretations and $T_P \uparrow n(I)$ may contains object expressions that are not compact, and as a consequence $T_P \uparrow \omega(I)$ as well. In order to apply T_P over the result of previous application $T_P \uparrow n(I)$, we first apply C to convert $T_P \uparrow n(I)$ into an interpretation.

Definition 5.10 Let $DB = (K, P)$. The operator T_P is *growing* if for each well-typed interpretation I, J, and M w.r.t. K, $I \sqsubseteq_p J \sqsubseteq_p M \sqsubseteq_p C(T_P \uparrow \omega(I))$ implies that $T_P(J) \subseteq T_P(M)$.

Definition 5.11 Let $T_1, ..., T_n$ be a sequence of operators. The *iterative powers* of the sequence with respect to an interpretation M are defined by

$$\begin{array}{ll} M_0 = M & \\ M_1 = C(T_1 \uparrow \omega(M_0)) & \text{if } C \text{ is defined} \\ M_2 = C(T_2 \uparrow \omega(M_1)) & \text{if } C \text{ is defined} \\ \vdots & \\ M_n = C(T_n \uparrow \omega(M_{n-1})) & \text{if } C \text{ is defined} \end{array}$$

Let $M_P = M_n$. The sequence of operators $T_1, ..., T_n$ is *local* if for each M and J such that $M \sqsubseteq_p J \sqsubseteq_p M_n$ implies $T_i(J) = T_i(J \sqcap M_i)$.

Locality means that each T_i is determined by its values on the subsets of M_i.

Lemma 5.1 Let K be a schema, P be a program stratified by $P = P_1 \dot{\cup} \ldots \dot{\cup} P_n$ and $T_{P_1}, \ldots, T_{P_n}$ the corresponding sequence of operators of P. Then this sequence is local.

Proof. Suppose that $T_{P_i}(J) \neq T_{P_i}(J \sqcap M_i)$ for some i. Let $\psi \in T_{P_i}(J)$. We prove $\psi \in T_{P_i}(J \sqcap M_i)$. Assume $\psi \notin T_{P_i}(J \sqcap M_i)$. (1) If $\psi \in T^1_{P_i}(J)$, then ψ is the result of applying some rule r in P_i to J. Since $\psi \notin T_{P_i}(J \sqcap M_i)$, the application of the rule uses a ground simple object expression φ in the body of r such that $\varphi \in J$ but $\varphi \notin J \sqcap M_i$. By Proposition 4.2, $J \sqcap M_i \sqsubseteq_p J$. So we have $\varphi \notin M_i$. Now that φ is used in the body of the rule, the defined symbol of φ must be in $D_1 \cup ... \cup D_i$. Since $J \sqsubseteq_p M_n$, there exists a φ' such that $\varphi \preceq \varphi'$ and $\varphi' \in M_n - M_i$. Thus, the defined symbol of φ', also the defined symbol of φ, is in D_j, for $j > i$. This is contrary to the stratification condition on P. (2) If $\psi \in T^2_{P_i}(J)$, then based on (1) it is straightforward that $\psi \in T_{P_i}(J \sqcap M_i)$. Therefore, $T_{P_i}(J) \subseteq T_{P_i}(J \sqcap M_i)$ The reverse inclusion is proven in a similar way. □

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

$$\pi_D(I) = \{\psi \in I \mid \text{the defined symbol of } \psi \text{ is in } D\}$$

Lemma 5.2 Let K be a schema, P a program stratified by $P = P_1 \dot{\cup} ... \dot{\cup} P_n$, M a well-typed interpretation w.r.t. K, D_1, ..., D_n the sets of defined symbols of the corresponding strata, and $T_{P_1}, ..., T_{P_n}$ the corresponding sequence of operators of P. Then $\pi_{D_1 \cup ... \cup D_{j-1}}(M) = \pi_{D_1 \cup ... \cup D_{j-1}}(T_{P_j} \uparrow \omega(M))$.

Proof. We prove by induction on i that $\pi_{D_1 \cup ... \cup D_{j-1}}(M) = \pi_{D_1 \cup ... \cup D_{j-1}}(T_{P_j} \uparrow i(M))$. The claim is obviously true for $i = 0$, since $T_{P_j} \uparrow 0(M) = M$. Since the defined symbols of P_j are not in $D_1 \cup ... \cup D_{j-1}$, the application of T_{P_j} to $T_{P_j} \uparrow i(M)$ does not change the extensions of the defined symbols of $D_1 \cup ... \cup D_{j-1}$. □

Corollary 5.1 Let K be a schema, P a program stratified by $P = P_1 \dot{\cup} ... \dot{\cup} P_n$, M a well-typed interpretation w.r.t. K, and M_i as defined in Definition 5.11 for $i = 1, ..., n$. Then for each i and j such that $1 \leq i \leq j \leq n$, $\pi_{D_1 \cup ... \cup D_i}(M_i) = \pi_{D_1 \cup ... \cup D_i}(M_j)$ and $M_i \sqsubseteq_p M_j$.

Proof. Direct from Lemma 5.2. □

Lemma 5.3 Let K be a schema, P a program stratified by $P = P_1 \dot{\cup} ... \dot{\cup} P_n$, $T_{P_1}, ..., T_{P_n}$ the corresponding sequence of operator of P. If each T_{P_i} $1 \leq i \leq n$ is defined on M_i as defined in Definition 5.11, then T_{P_i} is growing.

Proof. Let $I \sqsubseteq_p J \sqsubseteq_p M \sqsubseteq_p C(T_{P_i} \uparrow \omega(I))$. Let $\psi \in T_{P_i}(J)$. Then there is a rule in P_i of the form $A :- L_1, ..., L_n$ and a ground substitution θ such that $J \models \theta L_1, ..., J \models \theta L_n$ and ψ is a ground constituent expression of θA. For each θL_j, if θL_j is a comparison expression, then $M \models \theta L_j$ since it is interpreted in the standard arithmetic and set-theoretic interpretation in the term algebra. If θL_j is negative or has a constituent expression of the form $o[l \rightarrow \{o_1, ..., o_n\}]$ and there exists a rule such that $P[l \rightarrow \langle Q_1, ..., Q_n \rangle]$ is a constituent object expression of the head, then the defined symbols of L_j are in $D_1 \cup ... \cup D_{i-1}$. By Lemma 5.2 $\pi_{D_1 \cup ... D_{i-1}}(J) = \pi_{D_1 \cup ... \cup D_{i-1}}(M)$, so $M \models \theta L_j$. Otherwise, since $J \sqsubseteq_p M$, we still have $M \models \theta L_j$. Therefore $\psi \in T_{P_i}(M)$. □

Theorem 5.1 Let K be a schema, P a program stratified by $P = P_1 \dot{\cup} ... \dot{\cup} P_n$, M a well-typed interpretation w.r.t. K. If M_P is defined, then it is the unique minimal and supported model of P containing M.

Proof. Let P_0 be an empty set. We show, using induction on i, that M_i is a model of $P_0 \cup ... \cup P_i$ containing M. When $i = n$, M_n is then a model of P containing M.

For the basis, $M_0 = M$ is a 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 $P_0 \cup ... \cup P_{i+1}$ containing M. If M_{i+1} is not a model, then there exists a rule $r \in P_0 \cup ... \cup P_{i+1}$ of the form $A :- L_1, ..., L_n$ such that $M_{i+1} \not\models r$. If $r \in P_0 \cup ... \cup P_i$, then the defined symbols of A, L_1, ..., L_n are in $D_1 \cup ... \cup D_i$. Since $\pi_{D_1 \cup ... \cup D_i}(M_i) = \pi_{D_1 \cup ... \cup D_i}(M_{i+1})$ by Corollary 5.1 and M_i is a model of $P_0 \cup ... \cup P_i$, $M_{i+1} \models r$, which is a contradiction. If $r \in P_{i+1}$, and $M_{i+1} \not\models r$, then there exists a ground substitution θ, such that $M_{i+1} \models \theta L_1$, ... $M_{i+1} \models \theta L_n$, and $M_{i+1} \not\models \theta A$. For each j, if θL_j is a comparison expression, then $C(T_{P_{i+1}} \uparrow k(M_i)) \models \theta L_j$ for all $k \geq 0$. For this j, set $\alpha(j) = 0$. Here α is just a function which map each j in L_j to a number. If θL_j is negative or has a constituent expression of the form $o[l \rightarrow \{o_1, ..., o_n\}]$ and there exists a rule such that $P[l \rightarrow \langle Q_1, ..., Q_n \rangle]$ is a constituent object expression of the head, then the defined symbols of L_j is in $D_1 \cup ... \cup D_i$. So $C(T_{P_{i+1}} \uparrow k(M_i)) \models \theta L_j$ for all $k \geq 0$. For this j, also set $\alpha(j) = 0$. Otherwise, there exists a k such that $C(T_{P_{i+1}} \uparrow k(M_i)) \models \theta L_j$. Let $\alpha(j)$ denote this k and l denote $max\{\alpha(j) \mid 1 \leq j \leq n\}$. By Lemma 5.3, $T_{P_{i+1}}$ is growing. By Proposition 5.2, for $l' > l$, $C(T_{P_{i+1}} \uparrow l'(M_i)) \models \theta A$. Therefore, $M_{i+1} = C(T_{P_{i+1}} \uparrow \omega(M_i)) \models \theta A$, which is again a contradiction.

Next we prove M_n is minimal. Let N be a model of P containing M. We prove by induction on i that

$$\text{if } N \sqsubseteq_p M_i, \text{ then } M_i \sqsubseteq_p N. \tag{1}$$

For $i = 0$, it is part of the assumptions. Assume the claim holds for $i \geq 0$. We prove by induction on j that

$$C(T_{P_{i+1}} \uparrow j(M_i)) \sqsubseteq_p N. \tag{2}$$

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

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

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

$$
\begin{array}{lll}
T_{P_{i+1}} \uparrow k(M_i) & \subseteq M_i \cup T_{P_{i+1}}(N) & \text{by induction hypothesis} \\
C(T_{P_{i+1}} \uparrow k(M_i)) & \sqsubseteq_p C(M_i \cup T_{P_{i+1}}(N)) & \text{by Proposition 5.2 (2)} \\
 & = M_i \cup C(T_{P_{i+1}}(N)) & \text{by Proposition 5.2 (3) and (1)} \\
 & \sqsubseteq_p N & N \text{ is a model and by Proposition 5.3} \\
T_{P_{i+1}}(C(T_{P_{i+1}} \uparrow k(M_i))) & \subseteq T_{P_{i+1}}(N) & \text{by Lemma 5.3} \qquad (4) \\
T_{P_{i+1}} \uparrow (k+1)(M_i) & = T_{P_{i+1}}(C(T_{P_{i+1}} \uparrow k(M_i))) \cup T_{P_{i+1}} \uparrow k(M_i) & \\
 & \subseteq M_i \cup T_{P_{i+1}}(N) & \text{by induction hypothesis}
\end{array}
$$

Thus (3) holds for all k.

$$
\begin{array}{lll}
T_{P_{i+1}} \uparrow (j+1)(M_i) & = T_{P_{i+1}}(C(T_{P_{i+1}} \uparrow j(M_i))) \cup T_{P_{i+1}} \uparrow j(M_i) & \\
 & \subseteq M_i \cup T_{P_{i+1}}(N) & \text{by (4) and (3)} \\
C(T_{P_{i+1}} \uparrow (j+1)(M_i)) & \sqsubseteq_p C(M_i \cup T_{P_{i+1}}(N)) & \text{by Proposition 5.2 (2)} \\
 & = M_i \cup C(T_{P_{i+1}}(N)) & \text{by Proposition 5.2 (3) and (1)} \\
 & \sqsubseteq_p N & \text{by assumption and Proposition 5.3}
\end{array}
$$

So (2) holds for all j, we have $M_{i+1} \sqsubseteq_p N$. This proves (1) and concludes the proof. □

Theorem 5.2 Let K be a schema, P a program stratified by $P = P_1 \dot{\cup} ... \dot{\cup} P_n$, and $M = \{\}$. Then M_P is the unique minimal and supported model of P.

Proof. By Theorem 5.1, M_n is a minimal model of P. Let P_0 be an empty set. Now we prove by induction on i that M_i is a supported model of $P_0 \cup ... \cup P_i$ by showing $M_i \sqsubseteq_p C(\overline{T}_{P_i}(M_i))$ where $\overline{T}_{P_i}(K) = \cup_{j=0}^{i} T_{P_j}(K)$. 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 $M_{-1} = \{\}$. We now prove by induction that for all $j \geq 0$

$$M_j \sqsubseteq_p M_{j-1} \cup C(T_{P_j}(M_j)). \qquad (5)$$

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

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

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

$$
\begin{array}{lll}
T_{P_{j+1}} \uparrow k(M_j) & \subseteq T_{P_{j+1}} \uparrow \omega(M_j) & \text{by definition} \\
C(T_{P_{j+1}} \uparrow k(M_j)) & \sqsubseteq_p C(T_{P_{j+1}} \uparrow \omega(M_j)) & \text{by Proposition 5.2 (2)} \\
 & = M_{j+1} & \text{by definition} \\
T_{P_{j+1}}(C(T_{P_{j+1}} \uparrow k(M_j))) & \subseteq T_{P_{j+1}}(M_{j+1}) & \text{by Lemma 5.3} \qquad (7) \\
T_{P_{j+1}} \uparrow (k+1)(M_j) & = T_{P_{j+1}}(C(T_{P_{j+1}} \uparrow k(M_j))) \cup T_{P_{j+1}} \uparrow k(M_j)) & \\
 & \subseteq M_j \cup T_{P_{j+1}}(M_{j+1}) & \text{by (7) and induction hypothesis}
\end{array}
$$

Thus (6) holds for all k. Consequently

$$
\begin{array}{lll}
T_{P_{j+1}} \uparrow \omega(M_j) \subseteq M_j \cup T_{P_{j+1}}(M_{j+1})) & & \text{by (6)} \\
M_{j+1} = C(T_{P_{j+1}} \uparrow \omega(M_j)) & & \text{by definition} \\
= C(M_j \cup T_{P_{j+1}}(M_{j+1}) & & \\
\sqsubseteq_p M_j \cup C(T_{P_{j+1}}(M_{j+1})) & & \text{by Proposition 5.2 (3) and (1)}
\end{array}
$$

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

$$
\begin{array}{ll}
M_i \sqsubseteq_p M_{i-1} \cup C(T_{P_i}(M_i)) & \text{by (5)} \\
\quad \sqsubseteq_p C(T_{\overline{P}_{i-1}}(M_{i-1})) \cup C(T_{P_i}(M_i)) & \text{by induction hypothesis} \\
\quad = C(T_{\overline{P}_{i-1}}(M_{i-1}) \cup T_{P_i}(M_i)) & \text{by Proposition 5.2 (3)} \\
\quad \sqsubseteq_p C(T_{\overline{P}_{i-1}}(M_i) \cup T_{P_i}(M_i)) & \text{by locality} \\
\quad \sqsubseteq_p C(T_{\overline{P}_i}(M_i)) & \text{by definition}
\end{array}
$$

which concludes the proof. □

Let $M = \{\}$. Consider the program given in Example 3.3 again, we have

$$
\begin{array}{l}
M_1 = \{ann : person, tom : person, tom[parents \rightarrow \{ann\}]\} \\
M_P = M_2 = \{family(\{ann\}), family(\{ann\})[children \rightarrow \{tom\}]\} \cup M_1 = I
\end{array}
$$

which is the minimal and supported model of the program.

Definition 5.12 Let $DB = (K, P)$ be a database, the *declarative semantics* of the program P is given by its unique minimal and supported model M_P computed as above if it exists.

Given a program P, if the unique minimal and supported model exists, then it can be computed bottom-up using a finite sequence of fixpoints and therefore used as the intended semantics of P. There are two reasons why this minimal and supported model may not exist. One is that the inferred collection of object expressions is not consistent, therefore the operator C is undefined. Another reason is that some inferred object expressions are not well-typed, therefore the operator T_P is undefined. Both cases have to be checked at run time.

Definition 5.13 Let $DB = (K, P)$ be a database, Q a query of the form ?–$L_1, ..., L_n$, and θ a ground substitution for variables of Q. Assume M_P is defined. Then θ is an *answer* to Q based on P if $M_P \models \theta L_1, ..., M_P \models \theta L_n$.

6 Conclusion

In this paper, we have presented a typed deductive object-oriented database language which effectively integrates important features in deductive databases and object-oriented databases, such as powerful set representation mechanisms, object identity, complex objects, classes, class hierarchy, multiple inheritance with overriding and schema. None of the reported deductive languages has the same expressive power and data modeling power as ROL. We have developed a well-defined declarative semantics for ROL which provides a firm logical foundation to object-oriented paradigm.

We have implemented the ROL language as a single-user deductive object-oriented database system, which can be obtained through anonymous ftp from *ftp.cs.uregina.ca/pub/rol/*. Current implementation doesn't support nested sets and requires all classes to form a lattice. However, it supports schema queries and higher-order queries by allowing class and attribute variables. It also supports updates. The user can update schema, facts, and rules at run-time. All the examples used in this paper except Example 4.1 (nested sets) have been tested. Extensive examples can be found in the package. In the implementation, we effectively combine top-down and bottom-up strategies by alternating them and the performance is quite reasonable. The details about the implementation can be found in another paper [29].

Currently, we are working on various query optimization methods in order to enhance the performance of the ROL system. We intend to extend the language into a general purpose rule-base database programming language.

Acknowledgments

This work has been partly supported by the Natural Sciences and Engineering Research Council of Canada.

References

[1] S. Abiteboul. Towards a deductive object-oriented database language. *Data and Knowledge Engineering*, 5(2):263–287, 1990.

[2] S. Abiteboul and S. Grumbach. COL: A logic-based language for complex objects. *ACM TODS*, 16(1):1–30, 1991.

[3] S. Abiteboul and R. Hull. IFO: A formal semantic database model. *ACM Trans. on Database Systems*, 12(4):525–565, December 1987.

[4] S. Abiteboul and R. Hull. Data functions, datalog and negation. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 143–153, 1988.

[5] S. Abiteboul and P. C. Kanellakis. Object identity as a query language primitive. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 159–173, 1989.

[6] S. Abiteboul, G. Lausen, H. Uphoff, and E. Waller. Methods and rules. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 32–41, 1993.

[7] H. Ait-Kaci and R. Nasr. LOGIN: A logic programming language with built-in inheritance. *J. Logic Programming*, 3(3):198–215, October 1986.

[8] K.R. Apt, H.A. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundation of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann Publishers, 1988.

[9] M. P. Atkinson and O. P. Buneman. Types and persistence in database programming languages. *ACM Computing Surveys*, 19(2):105–190, 1987.

[10] R. Bal and H. Balsters. A Deductive and Typed Object-Oriented Language. In S. Ceri, K. Tanaka, and S. Tsur, editors, *Deductive and Object-Oriented Databases*, pages 340–359, Phoenix, Arizona, USA, December 1993. Springer-Verlag Lecture Notes in Computer Science 760.

[11] F. Bancilhon and S. Khoshafian. A calculus for complex objects. *J. Computer and System Sciences*, 38:326–340, 1989.

[12] M. L. Barja, A. A. A. Fernandes, N. W. Paton, M. H. Williams, A. Dinn, and A. I. Abdelmoty. Design and implementation of ROCK & ROLL: a deductive object-oriented database system. *Information Systems*, 20(3):185–211, 1995.

[13] C. Beeri, S. Naqvi, O. Shmueli, and S. Tsur. Set construction in a logic database language. *J. Logic Programming*, 10(3,4):181–232, April/May 1991.

[14] E. Bertino and D. Montesi. Towards a Logical Object-oriented Programming Language for Databases. In *Proc. Intl. Conf. on Extending Database Technology*, pages 168–183. Springer-Verlag, March 1992.

[15] O. P. Buneman, S. B. Davidson, and A. Watters. A semantics for complex objects and approximate answers. *J. Computer and System Sciences*, 43:170–218, 1991.

[16] F. Cacace, S. Ceri, S. Crepi-Reghizzi, L. Tanca, and R. Zicari. Integrating object-oriented data modelling with a rule-based programming paradigm. In *Proc. Intl. Conf. on Very Large Data Bases*, pages 251–261, 1990.

[17] M. Carey, D. DeWitt, and S. Vanderberg. A data model and query language for EXODUS. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, 1988.

[18] S. Ceri, G. Gottlob, and T. Tanca. *Logic Programming and Databases.* Springer-Verlag, 1990.

[19] W. Chen and D. S. Warren. C-logic for complex objects. In *Proc. ACM Symp. on Principles of Database Systems*, pages 369–378, 1989.

[20] C. J. Date. *An Introduction to Database Systems.* Addison Wesley, 6 edition, 1995.

[21] G. Dobbie and R. Topor. On the Declarative and Procedural Semantics of Deductive Object-Oriented Systems. *Journal of Intelligent Information System*, 4:193–219, 1995.

[22] D. H. Fishman, B. B., H. P. Cate, E. C. Chow, T. Connors, J. W. Davis, N. Derrett, C. G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M. A. Neimat, T. A. Ryan, and M. C. Shan. Iris: An object-oriented database management system. *ACM Trans. on Office Information Systems*, 5(1):48–69, January 1987.

[23] A. Heuer and P. Sander. The LIVING IN A LATTICE rule language. *Data and Knowledge Engineering*, 9:249–286, 1992.

[24] H. Ishikawa, F. Suzuki, F. Kozakura, A. Makinouchi, M. Miyagishima, Y. Izumida, M. Aoshima, and Y. Yamane. The model, language, and implementation of an object-oriented multimedia knowledge base management system. *ACM TODS*, 18(1):1–50, 1993.

[25] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of ACM*, 42:741–843, 1995.

[26] M. Kifer and J. Wu. A logic for programming with complex objects. *J. Computer and System Sciences*, 47:77–120, 1993.

[27] Won Kim. *Introduction to Object-Oriented Databases.* The MIT Press, 1990.

[28] C. Lecluse and P. Richard. The O_2 database programming language. In *Proc. Intl. Conf. on Very Large Data Bases*, pages 411–422, Amsterdam, The Netherlands, 1989.

[29] M. Liu and W. Yu. Implementation of the ROL rule-based object system. *In preparation*, 1996.

[30] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 2 edition, 1987.

[31] Y. Lou and M. Ozsoyoglu. LLO: A deductive language with methods and method inheritance. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 198–207, 1991.

[32] D. Maier. A logic for objects. Technical Report CS/E-86-012, Oregon Graduate Center, Beaverton, Oregon, 1986.

[33] Shamim Naqvi and Shalom Tsur. *A Logical Language for Data and Knowledge Bases.* Computer Science Press, 1989.

[34] F. Pfenning, editor. *Types in Logic Programming.* MIT Press., 1992.

[35] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. CORAL: Control, relations and logic. In *Proc. Intl. Conf. on Very Large Data Bases*, pages 238–250, 1992.

[36] M. Smyth. Power domains. *J. Computer and System Sciences*, 16:23–36, 1978.

[37] J.D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, 1988.