Deductive and Object-Oriented Databases
W. Kim, J.-M. Nicolas, and S. Nishio (Editors)
© Elsevier Science Publishers B.V. (North-Holland), 1990

Object Identity and Inheritance in Deductive Databases—an Evolutionary Approach

Carlo Zaniolo

MCC
3500 West Balcones Center Drive
Austin, Texas 78759
carlo@mcc.com

This paper proposes simple extensions to logic-based languages for deductive databases to support the key notions of object identity and object inheritance of object-oriented systems. Thus the paper shows how, given a logic program P and its minimal model $M(P)$, it is possible to construct a program P' and its minimal model $M(P')$ such that $M(P')$ is basically obtained from $M(P)$ by prefixing each fact of $M(P)$ with a unique identifier. This is accomplished through the use of $\mathcal{LDL}$ non-deterministic choice constructs (or alternatively through negation under the stable model semantics). This allow a user to view facts and predicates of ordinary logic programs as objects and refer to them through their identifiers—in both the extensional database and the intensional one. Then, syntactic shorthand conventions are introduced to facilitate reference based reasoning and access to objects. The result is a logic-based language where the ISA inheritance is fully supported along with the use of identifiers as surrogates for the actual objects. The proposed solution is firmly rooted in the theory of deductive databases and simple to implement as an extension of current deductive database systems.

1 Introduction

The topic of combining logic-based and object oriented systems has been among the favored subjects of recent research and topical discussion. The scope of this paper is rather modest, since it neither intenteds to serve as a survey of the field, nor as a general synthesis of previous solutions. Rather, this paper wants to provide a logical basis and lay down the technical foundations for the notions of object identity and object inheritance [KiLo] in deductive databases [GMN, Ull, NT89]. Even with this limited scope however, our task remains a challenging one; the difficulty of our undertaking is demonstrated by the long stream of papers on the problem of extending relational databases with the notion of object identity and inheritance. While this stream of research has produced several interesting proposals, it has not led to a clear convergence, a unifying synthesis or a wide agreement on some solution. As we now revisit the same problem in the new and richer framework of deductive databases we see that we are, on the one hand, faced with a more difficult technical challenge and, on the

other hand, provided with better tools to deal with them. Indeed we are faced with (i) the challenge of having to support a more powerful system (and for instance, having to deal with object identity and inheritance in the intensional database in addition to the extensional one of relational databases), and (ii) the opportunity offered by the greater expressive power of the (rule-based) deductive database language. The main contribution of this paper is the development of an evolutionary approach, whereby, taking full advantage of (ii), we define a new language supporting object identity and inheritance, as an extension of the Logical Data Language $\mathcal{LDL}$ [TsZa, NT89, Zani]. The new language will be called OIL, for Object, Inheritance and Logic. OIL is an extension of $\mathcal{LDL}$; so every valid $\mathcal{LDL}$ program is also a valid OIL program. Furthermore, the semantics of every new OIL construct is defined in terms of an equivalent $\mathcal{LDL}$ program.

While OIL captures the notions of object identity and inheritance (passive objects) it does not attempt to model concepts such as methods and object states found in the active object paradigm of programming languages. Thus the scope of this paper is different from that of previous papers where a full integration of logic programming and object oriented programming was sought. As a second remark, we observe that we use the $\mathcal{LDL}$ syntax for concreteness and because it is well-documented; yet ideas and techniques proposed here apply equally well to other Horn Clause languages for deductive databases [Meta, Ull]. In the paper, we will assume our reader familiar with the basic notions on logical languages [Lloy] and only explain constructs and notation that are particular to $\mathcal{LDL}$.

The main beacon in the design of OIL was represented by the language LOGIN [AiNa] whose appealing constructs for structured logical objects and lattice based inheritance we try to emulate—as others have done before us by either adopting or imitating LOGIN's constructs in other languages [AiLi, BNS]. Yet, OIL is semantically very different from LOGIN. In fact, unlike LOGIN, OIL comes with a (bottom-up) model theoretical semantics, supports explicitly the notion of object identity, and adopts the standard closed-world assumption for negation. Thus, semantically, OIL is a much clearer departure from LOGIN than a previous proposal [BNS], which focuses on providing a bottom up model theoretic semantics for the language. A first attempt to support the notion of object identity for LOGIN-like objects, is presented in the O-logic proposal [Maie]. The O-logic ideas were further developed, and various open issues resolved, in the powerful languages proposed in [KiLa, KiWu]. All these languages have a complex semantics, which uses an existential quantification for object identifiers in clauses, and a notion of model that is different and more complex than the standard one [Lloy]. The claim of this paper is that OIL outlines a simpler, more direct and more intuitive solution to the problem.

2 Object Identifiers

The key problem pertains to modeling the concept of object identifiers (IDs) within the logical framework. Previous authors have dealt with this problem in complex and often unsatisfactory ways. Our objective is to treat it in a simple way that is naturally rooted in the the semantics of logic-based languages.

Syntactically the new language, OIL, is an extension of $\mathcal{LDL}$. We will define a mapping $P \rightarrow O(P)$ that takes a program written in OIL and map it into an equivalent $\mathcal{LDL}$ program O(P). Then, we define the meaning of the OIL program P to be the same as the semantics of the LDL program $O(P)$ generated by this transformation. (LDL programs have a well-defined model theoretic and fixpoint semantics [NT89].) Pragmatically, therefore, we can view P as a short hand for $O(P)$.

The basic idea is to map each atom (predicate occurrence) in P,

$$\mathtt{p(X,...,Y)} \tag{1}$$

into the following atom of $O(P)$

$$\mathtt{Id : p(X,...,Y)} \tag{2}$$

where ':' is the infix symbol of a distinguished binary predicate. The left argument of ':' represents the identity of the object, while the second argument stores the value of the object. We will refer to binary ':' predicates as *tagget predicates.*

In conformity to what is done in many real systems, we will use positive integers for object IDs. Thus we assume that our OIL database consists of a set of facts each with the same ':' predicate symbol as (2). We will assume that there is a different ID integer value for each fact stored in the database. So, while the user sees in P facts such as:

```
father(anne, mark).
mother(mark, susan).
```

The actual fact base in $O(P)$ contains the two facts as the following ones:

```
10000 : father(anne, mark).
10001 : mother(mark, susan).
```

where the two integers are the respective IDs for the facts—the actual value of IDs is immaterial but no two facts should share the same value.

The next important question that we want to address concerns the generation of IDs for derived facts (intensional database). We will begin with a naive and incorrect solution, which we then refine into the final and ostensibly correct one.

Say that we have the following maternal-grandfather rule in P:

Example 2.1 *A rule to define maternal grandfathers.*

```
grandpa(Child, Gpa) ← mother(Child, Mother), father(Mother, Gpa).
```

To preserve the original semantics we will have to expand the goals in the tail to tagget predicates as follows:

```
grandpa(Child, Gpa) ← _ : mother(Child, Mother), _ : father(Mother, Gpa).
```

Observe that underscores are used in the left arguments of our tagget predicates, since the actual values of IDs are of no interest. We can now attend to the problem of generating IDs for the head predicate(s). Since we have decided that IDs should be integers we could try to write the following rule (`int(I)` is a recursive predicate that generates all positive integers a la' Peano):

```
I : grandpa(Child, Gpa) ← int(I),
                          _ : mother(Child, Mother), _ : father(Mother, Gpa).
```

Syntactically all our predicates are now in a tagget predicate form. Semantically, however, we are faced with a near disaster. The problem is the same that has beset previous approaches [Maie, KiLa, KiWu]: since variables in Horn Clauses are quantified universally, the minimal model for this program will replicate each pair `Child, Gpa` for each possible values of I (i.e., for each positive integer). What we need is instead one value for each pair `Child, Gpa`. Previous authors [KiLa, KiWu] have addressed this need with elaborate extensions which require new semantic framework based on an existential quantification for the new IDs tagging head objects (such as I in the previous example). Yet, all that is really required is to use $\mathcal{LDL}$'s *choice* construct to enforce a global uniqueness of IDs for tagget predicates. This will be accomplished through the use of an additional predicate, the unary predicate aux and of a global choice rule. Thus, for our example we have

Example 2.2 *The o-expansion of Example 2.1.*

```
I : grandpa(Child, Gpa) ← aux(grandpa(Child, Gpa), I),
                          _ : mother(Child, Mother), _ : father(Mother, Gpa).
aux(Op, Id) ←             int(Id), choice((Op), (Id)), choice((Id), (Op)).
```

Thus we use an additional distinguished predicate `aux(_,_)` to generate identity values for the derived predicate in the head of the rule. The binary aux predicate establishes a one-to-one correspondence between the value parts of derived objects (tagget predicates) and the positive integers that serve as object identifiers. Therefore, aux performs a global assignment of integer ID values to objects via the use of $\mathcal{LDL}$'s choice construct [KrNa, NT89]. A goal `choice((X),(Y))` ensures that a Functional Dependency, $X \rightarrow Y$, holds between the X-values and the Y-values in the model. A more detailed discussion of the finer points of the semantics of choice and its relationship with the stable semantics of programs is given in the Appendix. The previous example

illustrates how given a program P we obtain its object expansion $O(P)$. Observe that both P and $O(P)$ are $\mathcal{LDL}$ programs with a well-defined semantics.[1]

We can now investigate the relationship existing between the meaning of P and that of $O(P)$. Let $M(P)$ be the minimal model of P which defines its semantics; likewise, let $M(O(P))$ be the minimal model for $O(P)$. Let us disregard the auxiliary predicate aux. Then we see that $M(O(P))$ can be obtained from $M(P)$ by prefixing each fact of $M(P)$ with a unique integer identifier `I:`. In addition, P and $O(P)$ are query-equivalent, as explained more precisely next. If `?q` denotes a query P, then, `?_:q` will denote the corresponding query goal in $O(P)$. Moreover, let us assume now that underscores denote variables that are projected out in the output.[2] Then, we have that `?_:p` on $O(P)$ returns exactly the same result as `?p` on P. Thus for instance, a query

`?grandpa(X,Y).`

on Example 2.1 will return the same results as a query

`?_:grandpa(X,Y).`

on Example 2.2. Thus our transformation $P \rightarrow O(P)$ is one that preserves query equivalence. Therefore, we can rightfully view a program P as a short-hand notation of a program $O(P)$ where all the database and derived facts possess unique IDs.

We can now summarize our conclusions in a somewhat more formal way as follows. Say that an $\mathcal{LDL}$ program P is given, consisting of (a) a set of facts, (b) a set of rules and (c) a set of queries. The transformation of P into $O(P)$ will be called an *o-expansion* that consists of the following successive transformation steps:

Step 1: *Goal expansion.*

In each rule of P, replace replace each goal, `q`, by a goal `_:q`. Likewise change each query goal `?q` into `?_:q`.

Step 2: *Object identifiers in heads of rules, or unit clauses or facts.*

If `p` denotes the head of a rule, a unit clause or a fact, then replace `p` by `I:p`. Then add the goal `aux(I, p)` to the body of this rule. Likewise if `q` denotes a unit clause or a fact, replaces it by a rule with head `q` and body `aux(I,q)`.

[1] This statement needs more explanation in the case where P is an $\mathcal{LDL}$ program with negated goals in rules. Then $O(P)$ will not be stratified on the basis of the predicate symbol ':' since this appears both in the head and in negated goals of the tail. Yet, if P is stratified, we can similarly stratify $O(P)$ on the basis of the symbol name of the right argument of ':'. Then, the standard stratified semantics can be assumed for $O(P)$.

[2] This feature is not implemented in the current $\mathcal{LDL}$ system.

Step 3: *Object ID fixing:*

Add the the second rule in Example 2.2 to P.

For instance, the o-expansion of the program of Example 2.1 is given in Example 2.2. Thus, the following proposition follows immediately from the definitions.

Proposition: *Let P denote an $\mathcal{LDL}$ program consisting of facts, rules and query goals. If* `?p` *is a query goal of P, then* `?_:p` *is a query goal of $O(P)$. Then* `?p` *and* `?_:p` *return the same result.*

We have thus established that programs and their o-expansion are query equivalent. We can view this correspondence in different ways. From one viewpoint, we will regard P as short-hand of O(P). A second useful viewpoint consists in regarding $O(P)$ as an abstraction of an object-oriented implementation of P. For instance, in the current implementation of $\mathcal{LDL}$ [Ceta, Zani] both extensional and intensional objects are uniquely identified. Indeed, facts are assigned unique IDs when they are generated. These will change when objects are moved around or the database reorganized, but typically IDs of database objects remain unchanged in the database for considerable intervals of time. On the other hand, IDs for derived objects are generated dynamically at the time when a query is answered and they are soon after discarded. In many situations, IDs have a physical significance (e.g., they coincide with the addresses of memory cells containing the values of the respective objects) thus supporting a faster access to data.

3 Taking Advantage of Object Identities

In the previous section we have proposed an equivalent object-oriented interpretation to standard $\mathcal{LDL}$ programs, whereby a predicate `p(X,...,Y)` is regarded as a shorthand for an tagget predicate `Id:p(X, ..., Y)`. We will next allow our OIL programmer to mix both forms in the rules. Thus, we will allow

(1) predicate occurrences in the standard $\mathcal{LDL}$ form `p(X,...,Y)`,

(2) or in their tagget predicate form, `Id:q(X,...,Y)`, while

(3) query goals are allowed in both forms.

The o-expansion mapping of a program with tagget predicates remains the same as that defined early, with the exception that the goal expansion of Step 1 will leave tagget predicates unchanged. Then the meaning of a program P with tagget predicates is again defined by the meaning of $O(P)$.

Since object identities have now become available they can be used as references to object, i.e., logical surrogates to the object. For instance, suppose that in addition to our father and mother relation, there is a database relation as follows:

`person(Name, Age, Sex)`,

Then, we can write a new grandpa relation as follows:

Example 3.1 *Defining intensional relations between IDs.*

```
o_grandpa(Child, Gpa) ← granpa(Child_Name, Gpa_Name),
                        Child : person(Child_Name, _, _),
                        Gpa : person(Gpa_Name, _, _).
```

Thus, the o_grandpa is a relationship between object IDs. Object IDs can be stored in facts and used in rules (including updates rules, with all the benefits thereby issuing, e.g., in the preservation of existential constraints). While object IDs can be freely used in rules and derived predicates, their usage in the query goals should be restricted. [3]

Let us investigate next how object IDs can be used to enhance the expressiveness and naturalness of our language. Say that we want to define the record of a person to include information about the person's father:

Example 3.2

```
person_f(Name, Age, Sex, Father) ← person(Name, Age, Sex),
                                   father(Name, Fname),
                                   Father : person(Fname, _, _, _).
```

Thus, Father in person_f is now the reference to an object via its ID. A query such as "Find the age and sex of the grandchildren of a given person (denoted by his name Gf_name)" could be expressed as follows:

Example 3.3 *Age and sex of grandchildren of* Gf_name

```
gchildren(Gf_name, Age, Sex) ← person(Name, Age, Sex, F)
                               F : person(_, _, _, Gfid),
                               Gfid : person(Gf_name, _, _, _).
```

Now, we would like to blurr the distinction between an object and a reference to an object and to allow an equivalent expression of the previous example as follows:

[3] It has been traditionally suggested in the database field that users should never be able to see specific IDs values, since these have no logical significance and they can be changed by the system at any time. We feel that this is too restrictive, since, e.g., the ability of printing IDs of objects could be required for debugging purposes. Thus our preference is for allowing the printing of IDs values, but restricting users from giving ID values as part of a program or as an input. One possible way to implement that is modifying parsers and readers to return an error message whenever a user-supplied constant is used to address an tagget predicate. An even simpler solution consists of scrambling the values of IDs printed as to devoid these numbers of any logical or physical significance.

Example 3.4 *Age and sex of grandchildren of* Gf_name

```
gchildren(Gf_name, Age, Sex) ←
        person_f(Name, Age, Sex,
        F:person(_,_,_, Gfid:person(Gf_name,_, _, _))).
```

We thus have allowed the replacement of an object ID with the actual object. Again, we can regard the second rule as a short hand for the first rule that formally defines its semantic. Here all that is needed to implement the transformation $P \rightarrow O(P)$ is a trivial in-line expansion that transforms a nested structure of tagget predicates into a flat sequence (conjunct) of tagget predicates. Significant benefits in terms of conciseness and clarity follow from this notation of convenience. Thus if we had to check the presence of cycles of length 2 in our person_f relation we could use the following goal:

$$\texttt{Pid} : \texttt{person\_f}(\texttt{Name}, \texttt{Age}, \texttt{Sex}, \_ : \texttt{person\_f}(\_, \_, \_, \texttt{Pid} : \_))$$

while the following conjunct guarantees that the cycle is simple:

$$\texttt{Pid} : \texttt{person}(\texttt{Name}, \texttt{Age}, \texttt{Sex}, \texttt{F} : \texttt{person}(\_, \_, \_, \texttt{Pid} : \_)), \texttt{Pid!} = \texttt{F}$$

It is well-known that whenever one speaks of objects he must also mention the ISA inheritance—preferably in the same utterance [Beec]. It is easy to model the notion of inheritance in our framework. Suppose that in addition to the predicate

$$\texttt{person}(\texttt{name}, \texttt{Age}, \texttt{Sex})$$

we also have the predicate

$$\texttt{employee}(\texttt{Name}, \texttt{Age}, \texttt{Sex}, \texttt{EmP\#}, \texttt{Sal}).$$

Then we might want to include a statement to the effect that every employee is a person. Clearly we can add a value based implication such as

Example 3.5 *Implication without inheritance.*

$$\texttt{person}(\texttt{Name}, \texttt{Age}, \texttt{Sex}) \leftarrow \texttt{employee}(\texttt{Name}, \texttt{Age}, \texttt{Sex}, \texttt{EmP\#}, \texttt{Sal}).$$

But according to the semantics described above, this rule is interpreted as specifying the derivation of a new object person from the old object employee—thus two different IDs are generated for the two objects. Thus the following rule:

Example 3.6 *Find the salary of* Name.

```
father_sal(Name, Ftr_Sal) ←
        person_f(Name, _,_,_, F:employee(_, _, _, _, Ftr_Sal)).
```

to find the salary of a person's father, would fail, even when the father of our person is an employee. (Indeed, after the goal in the above rule is expanded into `person_f(Name, _,_,_,F), F:employee(_, _, _, _, Ftr_Sal))`, the search for an object `employee` with ID `F` fails, since `F` is uniquely associated with an object `person`, as per Example 3.3.) Here instead, we need to say that the given employee object and the derived person object share the same identity. This can be done by writing:

Example 3.7 *Implication with inheritance.*

$$\texttt{Id : person(Name, Age, Sex)} \leftarrow \texttt{Id : employee(Name, Age, Sex, EmP\#, Sal)}.$$

Then the previous rule will succeed for people whose father is an employee (the reader can verify that). This is not a small point, since it pertains to a feature that is at the core of the whole object oriented approach. This is the notion that a programmer should be able to write functions (or methods or procedures) on objects, and then have these work on later introduced specializations of said objects with no changes in the original program. Say for instance that our programmer wants to write a display procedure for objects of type person. Then he/she should define a predicate, say

```
display_person(PersonID)
```

that *given the ID* of a tagget predicate `person` does what it is supposed to do with its arguments. Now, by the mere addition of the above inheritance rule our old `display_person` procedure will also work for objects of type `employee`. In the next section we provide some syntactic sugaring to render the declaration of inheritance more convenient and expressive.

Example 3.7 shows a very common manifestation of the notion of inheritance. It describes the situation where objects are further specified with the introduction of additional attributes. Another, manifestation of inheritance is when objects are defined from others via restrictions on their argument values [AiNa]. For instance, we may want to say that a minor is a person with age less than 18, as follows:

Example 3.8 *Inheritance by Restriction.*

$$\texttt{Id : minor(Name, Age, Sex)} \leftarrow \texttt{Id : person(Name, Age, Sex), Age} < 18.$$

Again, observe that, if we have an object `minor`, with ID `Minid` we can use the goal `display_person(Minid)` to print it.

4 Inheritance and Attribute Names

We would like to support the capability of declaring an inheritance structure between objects and of assigning names to the arguments of both database and derived objects [AiNa]. This removes the problems associated with a strictly positional notation for predicate, including the need for remembering the position and meaning of each argument, and the need for listing all the arguments. Thus for instance, we would like the option to use the person_f predicate as follows (the sex argument is being ignored)

person_f(name => id(andy, doe), age => 25, father => Fid)

or as follows (argument order is immaterial)

person_f(name => id(andy, doe), father => Fid, age => 25)

and having these to share the same meaning of their positional equivalent:

person_f(id(andy, doe), 25, _, Fid)

In OIL this can be accomplished by the following declaration.

person_f(name, age, sex, father) ISA obj.

There are two consequences of this declaration. One is to associates name with the first argument in person_f, and also associates age, sex and father, respectively, with the second, third and fourth argument. The second effect of this declaration is to establish that the class person_f, is a specialization of the class obj; this is a statement of no semantic consequence since we assume that obj is the distinguished symbol denoting the top of our lattice—thus every tagget predicate is a specialization of obj. Using the argument names so defined, we can now rewrite the previous grandchildren rules as follows:

Example 4.1 *An equivalent rule for grandchildren*

```
gchildren(Gf_name, Age, Sex) ←
        person_f(age => Age, sex => Sex,
        father=> _:person(father => _:person(name => Gf_name))).
```

Our readers who are familiar with LOGIN will notice some obvious syntactic similarities. However, there are also clear semantic differences, since the use of argument names is here simply regarded as syntactic sugar—the meaning of this rule is defined in terms of its expansion into its positional equivalent. Arguments which are not used in the named version become underscores in the positional version of the rule.

Let us consider now a non-trivial specialization situation. Say that we want to define a student as a person_f attending a college, and record his/her school year and major. Then we can add a declaration such as:

student(college, year, major) ISA person_f.

This declaration defines a new object with three new named arguments in addition to the four arguments found in person. The first four arguments inherit their names from person, while the fifth, sixth and seventh are respectively named college, year and major. In addition to naming conventions, our ISA declaration results in the following inheritance rule:

I : person_f(V1, V2, V3, V4) ← I : student(V1, V2, V3, V4, _, _, _).

As a result of this rule, the success of goal

?student(name => id(andy, doe), age => 21, college => 'USC', major => arts)

Also implies the success of a goal:

?person_f(name => id(andy, doe), age => 21).

Furthermore, let us say that we have also a second ISA declaration:

employee(E#, Sal) ISA person_f.

Thus employee objects will have the first four arguments as persons, and the additional E#, and Sal arguments.

We may want to ask queries such as "Find an employee who is also a student in computer science". In order to support these sort of questions, we need to declare an object type that is both a student and an employee, as follows:

student_worker ISA employee, student.

This declaration results into a creation of a new object class with argument names inherited from both student and employee (nine arguments). Also it will produce a pair of inheritance rules to ensure that each student_worker is counted both as an employee and as a student. Now, to express the query, find art students who make a salary of $30,000 we can write:

?student_worker(name => X, major => art, Sal => 30000).

The generation of inheritance rules and the translation of argument names into positional notation involve rather straightforward rewriting operations.

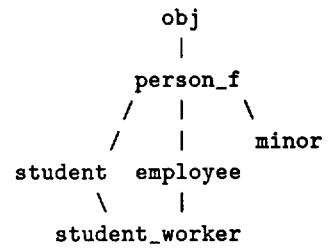
While an ISA declaration is all that is needed to support specialization through the addition of arguments, the situation where new objects are defined trough restrictions on argument values should be handled directly as described in Example 3.8. For instance, one could define minors from persons as follows:

Example 4.2 *Inheritance with restriction.*

```
minor ISA person_f.
Id : minor(name, Age, Sex, F) ← Id : person(Name, Age, Sex, F), Age < 18.
```

Then the inheritance declared by our previous ISA rules is shown in Figure 1, below. Observe that while it is always the case that there is only obj at the top of our ISA DAG, there normally exists no single bottom. This, example illustrates the power and simplicity of modeling inheritance by implication with ID sharing. Yet, this power could be used out of context, e.g., by letting two objects of totally different structure share the same ID.

Example 4.3 *The Inheritance Semilattice*



We will now give a second example illustrating the o-expansion for a situation involving employees and their department.

Example 4.4 *Employees, their departments and people in the management chain above them.*

```
dept(dname, loc) ISA obj.
emp(name, mgr, dept) ISA obj.
dept(dname=>shoes, loc=>austin).
partimer(WkHrs) ISA emp.
partimer(name=>id(j,brown),mgr=>id(ed,doe),
         dept =>_:dept(dname=>shoes),WkHrs=>25).
mgr(Eid:emp(mgr=>Mname), Mid:emp(name=>Mname)).
mgrStar(A,B) ← mgr(A,B).
mgrStar(A,C) ← mgrStar(A,B),mgr(B,C)
?mgrStar(_:emp(name=>X), _:emp(name=>Y)).
```

The semantics of this program is defined by its o-expansion of Example 4.5.

The o-expansion defines the semantics of our original query, and supplies the blueprints for its implementation. In the actual implementation, there will be a distinction between intensional predicates and extensional ones. For instance, partimer,

emp, and dept could be extensional predicates in our example. Then, there are three kinds or rules in any o-expansion. The first rule defines the uniqueness of IDs, and will realized by the built-in procedure that generates unique IDs in the actual implementation. Then we have rules defining extensional tagget predicates (the second, third and fourth rule in Example 4.5). These will be supported as integrity constraints (i.e., at update time). Finally, we have the remaining rules defining intensional tagget predicates that are compiled and then executed at query time as standard $\mathcal{LDL}$ rules.

Example 4.5 *$O(P)$ for P of previous example.*

```
aux(Op,Id) ← int(Id),choice((Op),(Id)),choice((Id), (Op)).
I:emp(V1,V2,V3) ← I:partimer(V1,V2,V3,V4).
I:dept(shoes, austin) ← aux(dept(shoes, austin),I).
I:partimer(id(j,brown), id(ed,doe),I1,25) ←
        aux(partimer(id(j,brown),id(ed,doe),I1,25), I),
        I1:dept(shoes,_).
I:mgr(Eid, Mid) ← aux(mgr(Eid, Mid), I),
        Eid:emp(_, Mname, _), Mid:emp(Mname,_, _).
I:mgrStar(A,B) ← aux(mgrStar(A,B)),I), _:mgr(A,B).
I:mgrStar(A,C) ← aux(mgrStar(A,C)),I), _:mgrStar(A,B), _:mgr(B,C).

?_:mgrStar(_I1, _I2), I1:emp(X,_,_), I2:emp(Y,_,_).
```

5 Conclusion

In this paper, we have shown that the concept of object ID and object type inheritance can be supported through simple extensions to current deductive database systems and languages.

We began by studying the problem of attaching unique object IDs to predicates, in both the extensional and intensional database, in a way that naturally mimics what is already done in actual implementations, including the $\mathcal{LDL}$ implementation. We proposed a solution to the problem in the form of a language OIL where the standard notation for a predicate such as p(X)) is considered a short hand notation for a predicate of the form I:p(X), where I denotes the ID value for this tagget predicate. We then showed how a one-to-one correspondence between the original predicate instances and integer IDs can be established using the declarative, nondeterministic choice construct of $\mathcal{LDL}$.

Then we showed that, through the explicit use of these object IDs in rules, it is simple to capture the basic notion of inheritance. Furthermore, syntactic sugaring (implementable through a simple preprocessor) allowed us to use objects IDs as syntactic surrogates for the actual objects in OIL programs. Finally, declarations of argument names and inheritance were used to enhance the naturalness and conciseness of the combined object/logic notation, bringing these to levels that approach those of languages as sophisticated as LOGIN [AiNa] and LIFE [AiLi].

The solution described in this paper is still a preliminary solution, with much room for further work and refinements. (For instance, the issue of finer control of duplicates in rules, has been left for future work, and so is the treatment of sets of objects.) Nevertheless, the paper has shown that a theoretically sound and practically appealing treatment of object identity and inheritance is within the scope of the science and the reach of technology of today's deductive databases. In fact, the semantics of the proposed extension is easily defined using standard $\mathcal{LDL}$ constructs. Furthermore, the implementation of this extensions is simple and can be accomplished easily and efficiently via simple extensions of current systems.

Instrumental in our treatment of object identity have been some recent advances in the treatment of non-determinism in the semantic foundations of deductive databases; these are briefly discussed in the Appendix.

Acknowledgments

The author expresses his appreciation to Roger Nasr, Shojiro Nishio and Ravi Krishnamurthy for their comments and helpful suggestions.

References

[AiLi] Ait-Kaci H. and P. Lincoln, "LIFE: A Natural Language for Natural Languages", MCC Technical Report ACA-ST-074-88, 1988.

[AiNa] Ait-Kaci H. and R. Nasr, "LOGIN: A Logic Programming Language with built-in Inheritance", *Journal of Logic Programming*, 3(3), p. 187–215, 1986.

[BNS] Beeri, C., R. Nasr and S.Tsur, "Embedding *psi*-terms in a Horn-clause Logic Language", *Procs. Third Int. Conf. on Data and Knowledge Bases—improving usability and responsiveness*, Jersualem, June 28-30, pp. 347-359, 1989

[Beec] Beech, D., "Position Statement on Object Oriented Database Systems and Knowledge Systems", First Int. Workshop on Expert Database Systems, Oct. 24-27, 1984, Kiawah Island, SC.

[Ceta] Chimenti D. et al., "An Overview of the LDL System," *Database Engineering Bulletin*, Vol. 10, No. 4, pp. 52–62, 1987.

[GMN] Gallaire, H.,J. Minker and J.M. Nicolas,"Logic and Databases: a Deductive Approach," *Computer Surveys*, Vol. 16, No. 2, 1984.

[GL88] Gelfond, M., Lifschitz, V., "The Stable Model Semantics for Logic Programming", Proc. 5th Int. Conf. and Symp. on Logic Programming, MIT Press, Cambridge, Ma, 1988, pp. 1070-1080.

[KiLa] Kiefer, M. and Lausen, G., "F-Logic:A Higher-Order Language for Reasoning about Objects, Inheritance and Scheme" *Procs. ACM SIGMOD Int. Conference on Management of Data*, pp.134-146, 1989.

[KiLo] Kim, W. and F.H. Lochosky (eds.), *Object-Oriented Concepts, Databases, and Applications* ACM Press, New York, NY, Addison-Wesley Publishing Co., 1989.

[KiWu] Kiefer, M. and J. Wu, "A Logic for Object-Oriented Programming (Maier's O-logic Revisited)", *Procs. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Programming*, 1989.

[KrNa] Krishnamurthy, R. and S. Naqvi, "Non-Deterministic Choice in Datalog", *Procs. Third Int. Conf. on Data and Knowledge Bases—improving usability and responsiveness*, Jersualem, June 28-30, pp. 416-424, 1989

[Lloy] Lloyd, J. W., *Foundations of Logic Programming*, Springer Verlag, (2nd Edition), 1987.

[Maie] Mayer, D. "A Logic for Objects", In *Workshop on Foundations of Deductive Databases and Logic Programming*, pp. 6-26, Washington D.C., August 1986.

[Meta] Morris, K. et al. "YAWN! (Yet Another Window on Nail!)", *Data Engineering*, Vol.10, No. 4, pp. 28–44, Dec. 1987.

[NT89] Naqvi,S. A., S. Tsur: *A Logical Language for Data and Knowledge Bases*, W. H. Freeman, 1989.

[TsZa] Tsur, S. and C. Zaniolo, "LDL: A Logic-Based Data Language," *Proc. of 12th VLDB*, Tokyo, Japan, 1986.

[SZ89] Saccá, D. and C. Zaniolo, "Stable Models and Non-Determinism in Logic Programs with Negation", submitted for publication, 1989.

[Ull] Ullman, J.D., *Database and Knowledge-Based Systems*, Computer Science Press, Rockville, Md., 1988.

[Zani] Zaniolo, C., "Design and implementation of a logic based language for data intensive applications" *Proceedings of the International Conference on Logic Programming*, Seattle, 1988.

A Choice Models and Stable Models

For simplicity of exposition, consider the following program:

Example A.1 *A simple program with choice*

```
q(X, Y) ← p(X, Y), choice((X), (Y)).
p(a, 1).
p(a, 2).
p(b, 1).
p(b, 2).
```

Following [NT89], we define the choice model of this program, M to consists of p-facts and q-facts as follows [NT89]: M is defined by first considering the minimal model N for the following program obtained by omitting the choice goal in the rule.

Example A.2 *The same program without the choice goal*

```
q(X, Y) ← p(X, Y).
p(a, 1).
p(a, 2).
p(b, 1).
p(b, 2).
```

The minimal model of this program is N = {p(a,1), p(b,1), p(a,2), p(b,2), q(a,1), q(a,2), q(b,2)}.

Then, the choice model M for the original program, is defined to be identical to the minimal model of the following program (after the extchoice facts have been left out):

Example A.3 *Constructing a choice model for the original program.*

```
q(X, Y) ← p(X, Y), extchoice(X, Y).
p(a, 1).
p(b, 1).
p(a, 2).
p(b, 2).
```

To complete our definition, we only need to explain how extchoice is computed. For this purpose, let F denote a set of functional dependencies on a relation R and $Q \subseteq R$. When Q satisfies every FD in F, we will say that Q is a *restriction of R (induced) by F*; Q will be said to be *maximal* when there is no other restriction of R by F that is a proper superset of Q.

Now, if N_p denotes the set of p-facts in N, then extchoice is defined to be a maximal restriction of N_p induced by $F = \{X \rightarrow Y\}$. In general, there a several such sets—non-determinism. For the case at hand, for instance, there are four maximal restrictions, as follows: {(a,1), (b,2) }, {(a,2), (b,2) }, {(a,2), (b,1) }, {(a,1), (b,1) } . Each of these sets produce a different choice model; for instance, the selection of the first pair leads to the following model:

M_1 ={p(a,1), p(b,1), p(a,2), p(b,2), q(a,1), q(b,2)},

while the choice of {(a,2), (b,2) } produces

M_2 ={p(a,1), p(b,1), p(a,2), p(b,2), q(a,2), q(b,2)}.

Consider now the situation where we have two choice goals in our rule.

Example A.4 *Establishing a one-to-one correspondence.*

```
q(X,Y) ← p(X,Y), choice((X),(Y)), choice((Y),(X)).
p(a,1).
p(b,1).
p(a,2).
p(b,2).
```

In this case $F = \{X \rightarrow Y, Y \rightarrow X\}$, and there are only two maximal restriction of N_q by F, as follows: {(a,1), (b,2) } and {(a,2), (b,1) } . These, respectively, yield M_1 and M_3 ={p(a,1), p(b,1), p(a,2), p(b,2), q(a,2), q(b,1) }.

Thus, the semantics of choice just described ensures that, given a relation p(X,Y) that is the cartesian product of two domains, we can construct a biinjection on these domains. This reflect the actual $\mathcal{LDL}$ implementation [Ceta], but is more restrictive than the original semantics [NT89] that allows any cascading of maximal functional restrictions. Thus, we do not accept as a valid choice model M_2' ={p(a,1), p(b,1), p(a,2), p(b,2), q(b,2)}, which could have been produced by first computing the maximal restriction by $X \rightarrow Y$ and then by $Y \rightarrow X$.

An alternative definition of the semantics of object IDs can also be obtained by using negation as follows:

Example A.5 *Using Negation and Stable Models*

```
q(X,Y) ←      p(X,Y), ¬others(X,Y).
others(X,Y) ← q(X,Y1), Y1 ≠ Y.
others(X,Y) ← q(X1,Y), X1 ≠ X.
p(a,1).
p(b,1).
p(a,2).
p(b,2).
```

This program can be assigned a precise meaning using the notion of *stable model* [GL88]. An equivalent constructive semantics can be obtained using the notion of

backtracking fixpoint [SZ89]. Thus, while computing the fixpoint of `q(X,Y)`, new `p(X,Y)`-facts can be added to the fixpoint, provided that no conflict exists with previous `X`-values and `Y`-values. It can be shown that the set of stable models of this program is identical to the set of choice models of the original program [SZ89].

In summary, we have defined two alternative but equivalent formal semantics which can be used to assign unique integer IDs to the objects in a minimal model of a logic program, via the use of the following rule:

Example A.6 *Counting the elements of the universe*

$$\texttt{aux(Op, Id)} \leftarrow \texttt{int(Id), choice((Op), (Id)), choice((Id), (Op))}.$$

While the semantics of this rule might seem intimidating, its actual realization in any implementation is trivial.