AN IMPLEMENTATION MODEL FOR REASONING WITH COMPLEX OBJECTS

Qiming CHEN
SM Research Institute
16 Bei Tai Ping Lu
Beijing 100039
CHINA

Georges GARDARIN
INRIA & UNIV. PARIS VI
SABRE Project
BP 105, 78153 LE CHESNAY
FRANCE
gardarinàmadonna.inria.fr

ABSTRACT

In this paper, we first propose a natural syntactical extension of DATALOG called NESTED_DATALOG for dealing with complex objects represented as nested predicates. Then, we introduce the token object model which is a simple extension of the relational model with tokens to represent complex objects and support referential information sharing. An implementation model of a NESTED_DATALOG program is defined by mapping it to the token object model which remains a straightforward extension of classical logical databases. Through this work, we can accommodate two basic requirements : The availability of a rule language for reasoning with complex objects, and the mechanism for mapping a complex object rule program to a relational DBMS offering a pure DATALOG rule language. In summary, the main contributions of the paper are the definition of a rule language for complex objects and the development of a technique to compile this complex object rule language to classical DATALOG.

1. INTRODUCTION

Database (DB) and Logic Programming (LP) technologies are moving fast towards a common destination. This holds true both from their common ancestor of mathematical logic and from the complementary benefits they can provide. From a database point of view, the major efforts made so far consist in the logical foundation given to relational DB [Gallaire84, Reiter84] and the extensions proposed to relational DB to support rules. These extensions yield DATALOG, a pure Horn clause rule language with a fixpoint semantics. DATALOG has been extended for handling functions, negation and set terms [Zaniolo 85] [Zaniolo 86] [Beeri 86] [Kuper 86] [Kuper 87] [Abiteboul87] [Gardarin87a] .

The handling of complex facts in logic programming is naturally integrated by using functions. It is possible to query these facts using predicates containing functional arguments. This facility makes logic programming more powerful and flexible than pure DATALOG based deductive databases, although typing and large predicate extensions are not well supported. Extending deductive database technology to handle complex objects has become a key issue for introducing it to engineering applications. However, to directly support database reasoning involving complex objects, the following problems still remain :
(a) The support of aggregation hierarchies with mutually nested tuples and sets, namely, the description of the references among objects in a logic framework.
(b) The consistency and complexity of the rule language semantics against the possible introduction of nested predicates, multilevel nested sets and so on.
(c) The support of information sharing, particularly, the link between a complex object rule language and an existing object management system adopting sharing by references.
(d) The availability of inferences at any given level of an aggregation hierarchy.

These problems have been investigated previously. Speaking roughly, they were tackled along the following two lines :
(1) Handling complex objects as nested functional terms [Zaniolo 85] [Zaniolo 86] [Beeri 86] [Abiteboul87], which more or less stays within the FOL world but has obvious limitations in predicating arbitrary objects and in supporting data abstraction and referential sharing. Most of these attempts allow the database to store complex facts in non normalized relation and introduce an extended relational algebra to deal with non flat relations.
(2) Defining a new theory for complex objects with an associated calculus and rule language. According to the approach described in [Bancilhon86], complex objects can be built from atomic objects by applying to them recursively a tuple and a set construct. Together with the subobject relationship $\leq$, which is essentially a containment relationship, the set of objects forms a

partial order lattice. This structure is used to define rule languages for reasoning about complex objects. However, the implementation model of such languages has not been properly developed. One of the goal of this paper is to propose such an implementation model on top of existing relational DBMSs.

Thus, we concentrate in this paper on the implementation of a rule language for manipulating complex objects. We first propose a natural extension of DATALOG for dealing with nested predicates, called NESTED_DATALOG. With NESTED_DATALOG, any predicate, nested or not, can have a database extension. To define an operational semantics and an implementation framework of NESTED_DATALOG, we introduce **the token object model**. It is a simple extension of the relational model with tokens to represent complex objects and support information sharing. Tokens are introduced to represent complex objects and to provide links among composing sub-objects. The multilevel configuration hierarchies of complex objects are represented in a level-independent fashion through the use of tokens. The transition of inferences at different levels of the object configuration hierarchy is made by passing tokens through flat predicates which represent the link between an object and its complex components, thus associating the abstract description of an object at a higher level and the more detailed description of it at a lower level.

Compared with previous approaches, the proposed framework for reasoning with complex objects has the following distinct characteristics :
(1) It is an operational model allowing system designers to build compilers for a complex object rule language. The proposed approach is to compile NESTED_DATALOG program in pure DATALOG, as techniques to compile pure DATALOG programs into optimized relational algebra programs are now well known (see [Gardarin88] for a detailed survey).
(2) Since the tokens provides links among different parts of a complex object, thus expands the aggregation hierarchy to a level-independent flat scheme, it may be implemented on top of classical relational systems. Such an implementation has already been considered by others; for example, building an object-oriented interface over a relational backend [Kiernan87] or compiling logic programming languages with nested terms into conventional relational algebra operations [Al-Amoudi87] require similar concepts and methodologies to that presented here.
(3) It allows the user to query any predicate, even those nested in other predicates. Thus, it allows the user to zoom in any particular level, while suppressing the details of other levels.
(4) It simplifies the semantics of LP with sets by introducing strong typing, which allows to map the multilevel nested sets and tuples to only one level tuples. Indeed, the token model does not refer any nested predicate or set.

The rest of this paper is organized as follows : NESTED_DATALOG is presented in section 2 as a natural extension of logic programming over complex objects. In section 3, the token object model is introduced as the implementation model of NESTED_DATALOG objects. Also, the mapping of complex terms to flat terms is described. Section 4 presents mapping strategies for rules with complex heads. It requires to specify special token generator predicates in DATALOG. Further discussions on the execution of the mapped rules are proposed.

2. SYNTAX OF NESTED_DATALOG

To introduce our rule language, let us see how complex objects would normally be dealt with as nested functional terms. As usual, $\{O_1,O_2...O_n\}$ represents the set of elements $O_1,O_2,...O_n$ and (O_1,O_2,O_n) is an ordered tuple constructed with elements $O_1,O_2,...O_n$. The example given below includes a complex fact `registration` with nested functional terms, which shows a natural syntax to represent complex facts with the tuple and set constructor. Note that we use both () and {} to enclose arguments of functions, in accordance with the argument type (i.e., set or ordered tuple). Additional parenthesis could be used, but we shall forget them in the sequel for clarity and simplicity. With classical approaches to complex objects [Tsur 86], predicates are top-level constructs which cannot be parameters of other terms or predicates; thus only `registration` is entitled to be a predicate symbol in this example. Courses, course, prof, students, student are function symbols. On the contrary, with our approach, every object as registration, courses, course, prof, students, student is a predicate and will be mapped to a relation as we shall see below.

```
registration (cs, 1987, courses {
  course ( 231, db, prof (smith, male, 36),
           students { student (john, 17),
                      student (jan, 18),
                      student (hull, 20)} ),
  course ( 171, os, prof (smith, male, 36),
           students { student (lee, 18),
                      student (jan, 18),
                      student (hull, 20)} ),
  course ( 281, ai, prof (smith, male, 36),
           students { student (john, 17),
                      student (lee, 18),
                      student (hull, 20)} ) } ).
```

Suppose we have queries such as :
(a) Find the set of student names y professor x (name) teaches on course c (course number).
(b) Find student lee's age.
To express and answer these queries in a system supporting functions, one must start from the top predicate `registration`, then go all the way down to the points where the required information can be found. Even for a very simple query such as (b) it could not be processed directly since `student(lee,18)` has been treated as a **functional term**, thus cannot be taken as a **predicate** in the same logical universe. The rule language we are now going to define will explicitly consider nested predicates and support direct queries and rules.

We shall extend the above natural syntax to a rule language for complex objects, called NESTED_DATALOG. We now define the syntax of NESTED_DATALOG. As COL [Abiteboul87], NESTED_DATALOG is a strongly typed language. We start with sets of atomic objects which are the basic types. As usual, we use the following symbols :
(1) Constants : a, b, c, ... filler '-';
(2) Variables : x,y,z, ...;
(3) Type predicates : P, Q, R ...;
(4) Comparison predicates : $=, <, \leq, >, \geq, \neq$;
(5) Logical connectors : ', ' (i.e., and), $\leftarrow$ (i.e., is implied by).

A **simple term** is defined as being either a variable or a constant. For example, 1 and x are terms. Among the type predicates, there is a distinguished subset of unary predicates called **simple types.** We consider that all constants are typed according to a simple type. Typing constants corresponds to the classical assignation of domains in relational databases.

The novelties of the language are to support nested predicates (which indeed exists already in PROLOG, LDL1 or COL, but in that languages, internal predicates are considered as functions) and to be strongly typed. Complex types and Complex terms may be defined as follows.

Definition 2.1 : Complex type
A complex type is defined recursively as :
(a) Any type predicate is a complex type.
(b) If $p_1, p_2, \ldots p_n$ are complex types then $p(p_1, p_2, \ldots p_n)$ is a complex type (called a tuple type).
(c) If p' is a type then $p\{p'\}$ is a complex type (called a set type).

Definition 2.2 : Complex Term
A complex term is defined recursively as :
(a) If $t_1, t_2, \ldots, t_n$ are simple or complex terms and p is a n-place type predicate, then $p(t_1, t_2, \ldots, t_n)$ is a complex term (called a tuple term).
(b) If $t_1, t_2, \ldots, t_n$ are simple or complex terms and p is a unary type predicate, then $p\{t_1, t_2, \ldots, t_n\}$ is a complex term (called a set term).

A term t tagged explicitly with a type symbol p, as p(t), is called a strongly typed term. The above definition requires complex terms to be strongly typed. For example, a typed term p {q{x}} is a legal NESTED_DATALOG term, but neither {{x}} nor p({x}) are legal with NESTED_DATALOG since not strongly typed.

Note that a complex term may be represented using a graphical tree representation similar to that of [Bancilhon86], as illustrated with the registration complex fact in figure 1. The () nodes represent ordered tuples and the {} node represents sets. Leaves are simple terms. A depth-first traversal of the tree generates the complex fact. Figure 2 gives a representation of the complex type corresponding to the registration complex object. Note that leaves are now corresponding to atomic types (i.e., domains).

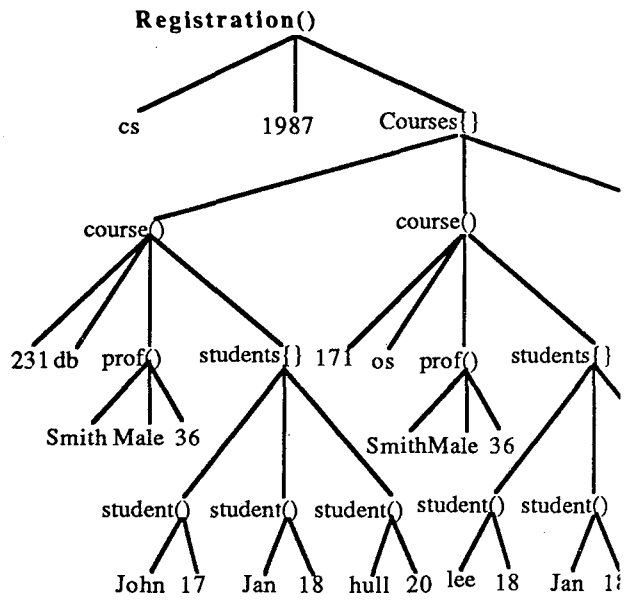


Figure 1 : A graphical representation of the registration complex object (partial view).

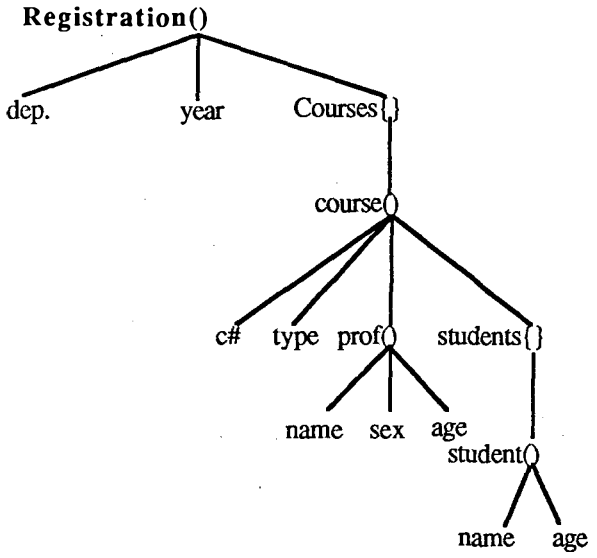


Figure 2 : A graphical representation of the registration complex type

With the notion of complex term, one can construct a set of syntactically well formed formulas (in short, formulas) as follows :

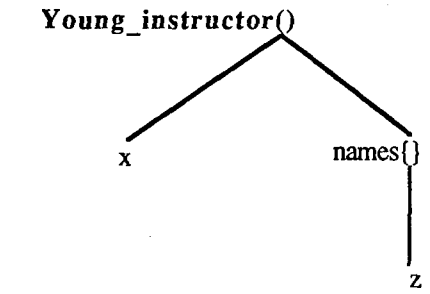
Definition 2.3 : Formula
(1) If p is a type predicate or a comparison predicate and $t_1, t_2, \ldots, t_n$ are complex terms, then $p(t_1, t_2, \ldots, t_n)$ is an atomic formula.
(2) Atomic formulas are well formed formulas.
(3) If F1 and F2 are well formed formulas, so are F1,F2.

Definition 2.4 : Rule
*A **rule** is defined as **head** ← **body**, where the body is a formula and the head is an atomic formula.*

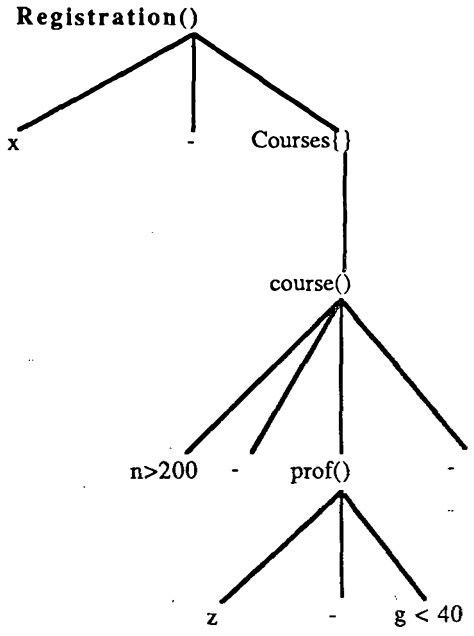
Definition 2.5 : Program
*A **program** is a finite set of rules.*

Definition 2.6 : Query
*A **query** is a rule without a head, denoted as **?body**.*

For example, using the registration complex type, one may define the rule :
young_instructor (x, names{z}) ←
registration (x, -, courses {course (n, -, prof (z, -, g), -)}), n > 200, g < 40
where the young_instructor predicate as defined above gives the set of names of all the young professors (age < 40) instructing courses such that (course_number > 200) in the department x. A rule can also be graphically represented as in figure 3, as a mapping of complex types to a new complex type.



(a) Complex head



(b) Complex body

Figure 3 : A graphical representation of a rule

3. MAPPING COMPLEX TERMS TO FLAT TERMS

As mentioned above, NESTED_DATALOG is implemented using DATALOG on top of an enhanced relational model, called the **token object model.** In this paper, for reasons of conciseness, we do not define the semantics of NESTED_DATALOG. Indeed, the semantics is the most natural one; for instance S{1,2,3} means that 1, 2 and 3 are elements of a set of type S. The query ?S{x}, x≥2 retrieves all elements of S greater than 2. Several papers already tried to define the semantics of a complex object rule language [Bancilhon86], [Abiteboul87], [Kuper87]. Our aim is not to define formally the language semantics, but to

study an implementation on top of a relational DBMS. The reader may consider that the language semantics is defined through its implementation, although it could be defined in other ways (e.g., by considering nested predicates as functions and by completing the rule program with unesting rules). Thus, we intend to precisely define the mapping of NESTED_DATALOG programs to the DATALOG implementation model for their FOL based evaluation.

In the current section, we describe the representation of tuples and sets with the token object model. We also present the mapping from NESTED_DATALOG terms to token object model terms.

3.1 Tuple and set Mappings

In general a complex object is constructed progressively from more primitive lower level objects; thus, it has a configuration hierarchy represented as a graph in the above examples. The representation of a complex object may refer other objects or be referenced by still more complex objects. To identify complex objects properly and uniquely is important both for maintaining the system consistency and for supporting the information referential sharing. Using surrogates [Maier83] or tokens [Woelk86] to represent abstractly an object with complex structure is beneficial from both implementation point of view and conceptualization point of view. In short, a token is an atomic object of a predefined type which uniquely identifies a complex object and stands for the whole configuration structure of the object. For simplicity and convenience, an atomic object needs not be associated with an additional token: It is identified by its own value.

The token object model is an implementation model for complex objects. It is composed of **flat relations** (i.e., 1NF relations) with **tokens.** To map complex objects to flat objects, we define the following three mappings.

Definition 3.1: Token Mapping
*The Token Mapping τ maps an object **O** to its token τO (also denoted *o) which is an atomic object of type token identifying the object **O**.*

Thus, mapping τ simply gives the token of an object. This token belongs to an atomic type, the token type. As stated above, for simplicity and convenience, any atomic object O satisfies $\tau O = O$. As a token is an atomic object, we have $\tau(\tau O) = O$.

Definition 3.2: Tuple Mapping
The Tuple Mapping ρ maps a type p tuple object $O = P(O1,...On)$ to a token object ρO defined as :

$$\rho O = {}^{*}P(\tau O,\tau O_1, ..., \tau O_n).$$

Definition 3.3: Set Mapping
The Set Mapping σ maps a type p set object $O = P\{O1,...On\}$ to a set of token object σO defined as: $\sigma O = \{{}^{+}P(\tau O,\tau O_1), ..., {}^{+}P(\tau O,\tau O_n)\}$.

The three proposed mappings are rather straightforward : τ gives the token of an object, ρ replaces a tuple by a tuple of tokens containing the tuple reference and references in place of complex objects, σ expands a set as a relation giving the token of the set for each element of the set represented by its token (or its value, which is also the token, in case of an atomic object). In the following, we shall denote a token with a minuscule letter preceded by a star for clarity. Also note that predicates resulting from tuple mapping are prefixed by a star while predicates resulting from set mapping are prefixed by +. This will be useful later in this paper.

3.2 Mapping Complex Terms to Token Object Terms

The mapping of facts from NESTED_DATALOG to the token object model is essentially a replacement of the complex objects by the corresponding token objects. It may be performed in a bottom-up manner, starting from the innermost predicates. In general, each tuple object p(T) generates a token predicate $^{*}p(\tau T,T)$ (i.e., tuple mapping is applied) and is replaced by the token τT in the remaining configuration hierarchy. Each set object $p\{S\}$ where $S = \{s1,...sn\}$ generates a set of token predicates $^{+}p(\tau S,s1)$, ... $^{+}p(\tau S,sn)$ (i.e., set mapping is applied) and is replaced by the token τS in the remaining configuration hierarchy.

More generally, let **x** be a complex term of NESTED_DATALOG and **u** be a set of DATALOG terms, the mapping :

$$\omega_{\mathcal{B}} : x \rightarrow u$$

is called a **term rewriting transformation.** Term rewriting may be defined as follows.

Definition 3.4 : Term rewriting
Set $\Omega_{\mathcal{B}}$ of term rewriting transformations defined recursively as :

(a) $v \rightarrow v \in \Omega_B$ *where v is a constant, a symbol or a filler;*

(b) if $v_1 \rightarrow u_1, ..., v_n \rightarrow u_n \in \Omega_B$,

then $P(v_1, ..., v_n) \rightarrow {}^*P(*p, u_1, ..., u_n) \in \Omega_B$;

(c) if $v \rightarrow u \in \Omega_B$, *then* $P\{v\} \rightarrow {}^+P(*p, u) \in \Omega_B$

*where *p's are token variables.*

A term rewriting is valid if all the token variables introduced are distinct.

From a complex term, term rewriting can be applied in a bottom up fashion, step by step unquoting the expression from the inner level to the outer level, while generating lists of single level predicates from the left to the right. The final list of predicates is yielded by the concatenation of all lists.

We now give an example.
Term rewriting is applied to :
registration(x, -,courses{course (n, -, prof (z, -, g), -)}).
The stepwise results of complex term rewriting (innermost first) is :

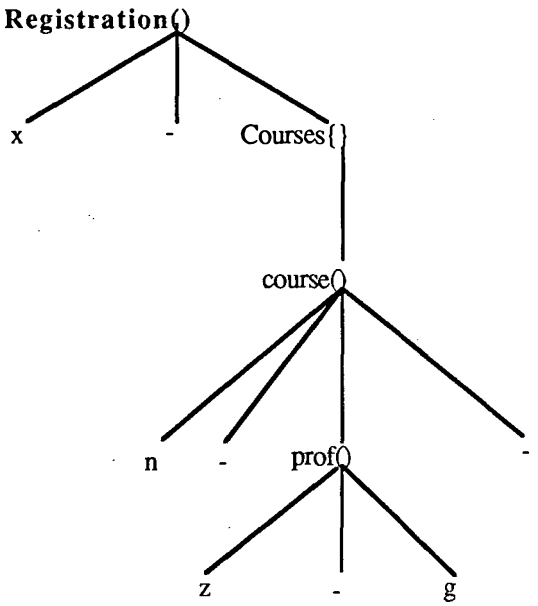
step 1 :
registration (x, -, courses { course (n,-,*p,-) })
with the following DATALOG predicate generated:
*prof(*p,z,-,g)

step 2 :
registration (x, -, courses {*c })
with the following DATALOG predicate generated:
* course (*c,n,-,*p,-)

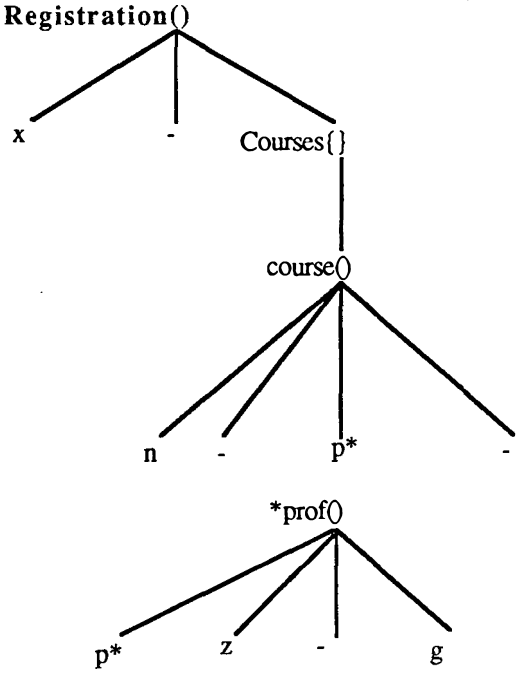
step 3,4 :
registration (x, -, *s)
with the following DATALOG predicates generated:
+courses (*s,*c)
*registration (*r,x, -, *s)

The four steps are illustrated in figure 4. Note that the transformation is a progressive reduction of the complex object tree from the leaves: Complex object sub-trees are progressively extracted and replaced by tokens which refer to them. At the last step, we introduce a token variable (i.e., *r) to be consistent with possibly other rules in which registration can be nested. This could be avoided if registration was known as being in all rules a top level predicate.

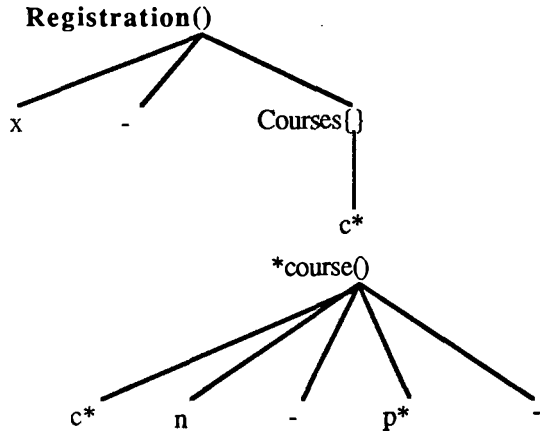
Step 1 :
The complex term :



is rewritten as :



Step 2 :
The remaining complex term **Registration()** is rewritten as :



Step 3 :
The remaining complex term **Registration()** is rewritten as :

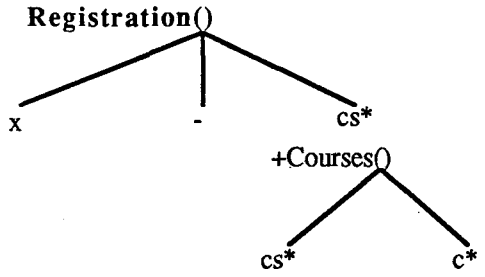


Figure 4 : Complex term rewriting as a tree reduction

Simple terms are directly rewritten without transformations. Thus, the NESTED_DATALOG rule with simple head :

young_instructor (x, z) ←
registration (x, -, courses {course (n, -, prof (z, -, g), -)}), n > 200, g < 40
is mapped to :

young_instructor (x, z) ←
*registration (*r,x, -,*s), $^{+}$courses (*s,*c),
*course (*c, n, -, *p, -), *prof (*p, z, -, g),
n > 200, g < 40.

which is pure DATALOG. Complex term rewriting is sufficient to map rules with simple heads. We study the case of complex heads in the next section.

4. COMPLEX HEADS AND FURTHER MAPPINGS

4.1 Typing with NESTED_DATALOG

NESTED_DATALOG is a strongly typed language. The mapping from a NESTED_DATALOG fact or rule to the set of DATALOG clauses with tokens essentially consists in the conversion of objects to flat tuples connected by tokens and the conversion of types to predicates, therefore transforming all the nested terms into flat predicates. Indeed, to have a NESTED_DATALOG fact or rule to be computable based on the token object model requires the converted set of DATALOG clauses to be fully predicated, and to have the resulted set of DATALOG clauses to be fully predicated requires that the original NESTED_DATALOG fact or rule to be **strongly typed.**

For example, a typed NESTED_DATALOG term P(Q{ R {x} }) can be mapped using the term rewriting definition to the conjunction of DATALOG clauses with tokens *P(*q),$^{+}$Q(*q,*r),$^{+}$R(*r,x) . However, neither {{x}} nor P({x}) can be mapped to a set of fully predicated DATALOG clauses with tokens, through the mapping rules given above, since they are **not strongly typed.** This is why they are not legal NESTED_DATALOG terms and cannot be evaluated on the DATALOG framework with tokens.

4.2 The problem of complex heads

Using complex heads in rules is necessary to restructure complex objects in other complex objects. However, mapping complex heads to flat predicates turns out to be a tedious task. Let us give three simple examples to show the problems (denoted (a) and (b) below) which arise :

(1) Tupling :
P(x,Q(y)) ← B(x,y)
Complex term rewriting applied to the head leads to the rule :
*P(x,*q),*Q(*q,y) ← B(x,y)

(2) Nesting :
P(x,Q{y}) ← B(x,y)
Complex term rewriting applied to the head leads to the rule :
*P(x,*q),$^{+}$Q(*q,y) ← B(x,y)

(3) Reconstruction :
R(S(x),y) ← P(x,Q{y})
Complex term rewriting applied to the body and the head yields the rule :
*R(*p,y),*S(*p,x) ← P(x,*q),$^{+}$Q(*q,y).

(a) Token variables appear in the transformed head which are not defined in the body. Thus, tokens have to be generated to identify either tuples or sets. The case of tupling is simple : It is possible to generate the head token by generating a token for certain tuples in a body predicate. To generate tokens, we introduce special predicates denoted TOKENn(*t, t1,t2,...,tn) which label every ordered tuple (t1,t2,...tn) of n attributes with a

token *t. Two identical tuples are labeled with the same token. For instance, in example (1), we can rewrite the transformed rule as :

*P(x,*q),*Q(*q,y) ←
B(x,y), TOKEN2(*q,x,y)

In the case of sets, the generation of tokens is more difficult as the set token must be identical for all elements of a given set. Thus, tokens have to be generated according to the nesting variables. For instance, in example (2), we can rewrite the transformed rule as :

*P(x,*q),$^{+}$Q(*q,y) ← B(x,y), TOKEN1(*q,x)

(b) Transformed rules are not anymore DATALOG rules as they include several predicates in the head. However, they can be split in several Horn clauses as tokens keep the link between the separately generated heads. For instance, example (1) will give the two DATALOG clauses :

*P(x,*q) ← B(x,y), TOKEN2(*q,x,y)

*Q(*q,y) ← B(x,y), TOKEN2(*q,x,y)

while example (2) will give the two DATALOG clauses:

*P(x,*q) ← B(x,y), TOKEN1(*q,x)

$^{+}$Q(*q,y) ← B(x,y), TOKEN1(*q,x)

Let us now define more precisely the concept of token generator and how we propose to use token generators to handle complex heads.

Definition 4.1 : Token generators
*Special system predicates denoted TOKENn(*t,t1,t2,...tn) which generates a token for each tuple (t1,t2,...tn), two tuples having same values giving the same token.*

Using token generators, it is possible to map NESTED_DATALOG rules to DATALOG rules with tokens. The mapping can be sketched as follows :

(i) Rewrite each complex term in the rule as simple terms with tokens using term rewriting as defined in section 4.

(ii) For each token variable which appears in the head and not in the body, add a token generator to the rule body. The arguments of the token generator predicate are the generated token variable followed by the ordered list of variables which appear in a head predicate prefixed by a star (*) containing the token variable. It may appear that no variables appear as arguments ; in that case, a special new token is simply generated to identify a set value. It may also appear that one wants to label with a token a tuple which already exists labeled with a token in the rule body. In that case, the token generator is not necessary; it is sufficient to rename the token variable in the head as in the rule body.

(iii) Split multiple head rules by duplicating the rule body for each simple head.

Using the previous mapping rules to generate the TOKENn predicates, NESTED_DATALOG programs can be compiled in DATALOG programs. When running the DATALOG program a few problems may appear which are examined below.

4.3 Variable Scope of NESTED_DATALOG and Extended Reasoning

In pure DATALOG, there is only one sort of variables, the atomic variables. In certain DATALOG-like languages with sets, such as LPS [Kuper87], there are two sorts of variables, the atomic variables and the set variables. In NESTED_DATALOG, a variable can represent a complex term. In principle, there is no need to restrict the scope of variables; they can be atomic, or mutually nested tuples and sets of any order. In a NESTED_DATALOG query, if a variable is an atomic one, the answer to the query under the DATALOG with token program will return the actual value as the substitution to the variable. However, if a variable is a non-atomic one, the normal query processing terminates with a token for the variable returned.

For example, let us consider the NESTED_DATALOG fact :

R (c, Q{P(a, 1), P(b, 2)})

which is transformed to the following clauses of DATALOG with tokens :

*P(*p1,a,1)
*P(*p2,b,2)
+Q(*q1,*p1)
+Q(*q1,*p2)
*R(*r1,c,*q1)

The following NESTED_DATALOG query with an atomic variable x :

S(x) ← R (c, Q{ P(x, 1)})

is transformed into the query of DATALOG with tokens

S(x) ← *R(*r,c,*q),+Q(*q,*p),*P(*p,x,1)

and has the answer x = a.

However, the following NESTED_DATALOG query with a set component variable x :

S{x} ← R (c, Q{x})

is transformed into the query of DATALOG with tokens

+S(*s,x)←R(*r,c,*q),+Q(*q,x),TOKEN0(*s)

and has the answer x = {*p_1, *p_2}, which is just a set of tokens. If tokens are made transparent to users such an answer may not be considered as the appropriate termination of the query processing.

To solve this problem we have developed the notion called **Extended Reasoning**, that is, if the query processing turns out to an answer with tokens involved, those tokens will be replaced step by step by their associated values, until no token occurs in the derived answer. Thus the above query will have the final answer : x = {P(a, 1), P(b, 2)}.

The extended reasoning is handled automatically by the system and need not be expressed explicitly in the user's program. It includes the following steps (we suppose that tokens form a special type of constants which can be recognized by the system; remember that every constant is typed) :
(1) For each token *t occurring in the answer, search the token predicates *g(*t,u), where g is any predicate symbol. Then replace *t by u.
(2) If u is still a token or a construct including tokens, do (1) again for each token appearing in u;
otherwise, return g(u) as the replacement of *t in the final answer.

5. CONCLUSION

In this work, we try to accommodate two basic requirements : A natural style of complex object rule language which represents hierarchical structures and a first order logic framework on which the consistent reasoning with complex objects is performed. Thus, from theoretical point of view, we propose a natural non-1NF rule framework that can be mapped to the classical 1-NF rule framework. We formally and precisely defined the necessary mappings. From a practical point of view, that demonstrates the possibility to support engineering applications on top of relational systems. The performance issue is a question for further studies.

In summary, we started with describing a natural rule language called **NESTED_DATALOG** for dealing with complex objects and supporting the deductive retrieval and reconstruction of nested predicates. With introducing the Token Object Model, we show that it is possible to map NESTED_DATALOG to DATALOG on top of a relational DBMS supporting the **token** concept. By starting with NESTED_DATALOG and studying how hierarchically structured facts and rules can be converted to a set of "flat" clauses for the FOL-based reasoning, interesting properties of NESTED_DATALOG programs can be inferred. For example, properties such as safety and stratification could be studied. To go further along this line will be a significant future work.

REFERENCES

[Abiteboul 87] S. Abiteboul, S. Grumbach, "Une Approche Logique de la Manipulation d'objets Complexes", 3e Journées BD3, Port Camargue, Mai 1987.

[Al-Amoudi87] Al-Amoudi S., Harper D., "On Compiling Logic Programming Languages into Conventional Relational Algebra Operations", Internal Report, University of Glasgow, UK, 1987.

[Apt86] Apt K.R., Blair H., Walker A. ,"Towards a Theory of Declarative Knowledge", IBM Research Report RC 11681, April 1986, in Proc. Workshop on Foundations of Deductive Databases and Logic Programming, Washington D.C., pp. 546,629, 1986.

[Bancilhon 86] F. Bancilhon and S. Khoshafian, "Objects Calculus", Proc. PODS, 1986.

[Beeri 86] C. Beeri et.al, "Sets and Negation in a Logic Database Language LDL1", MCC Tec. Rep, 1986.

[Chen 86] Q. Chen, "A Rule-based Object/Task Modeling Approach", Proc. of ACM-SIGMOD 86, USA.

[Chen 87] Q. Chen, "An Extended Object-Oriented Database approach", Approach", Proc. of COMPSAC 87, Japan, 1987.

[Gardarin 87a] G. Gardarin, "Magic Functions: A Technique to Optimize Extended Datalog Recursive Programs", proc. VLDB 13, London, 1987.

[Gardarin 87b]G. Gardarin, E. Simon, "Les Systèmes de Bases de Données Déductives", TSI, Dunod Ed., March 1988, in French.

[Gardarin88] G. Gardarin, P. Valduriez, "Principles and Algorithms of Relational Database Systems", Book, Addison-Wesley, 1988.

[Gallaire84] Gallaire H., Minker J., Nicolas J.M. : "Logic and databases : a deductive approach", ACM Computing Surveys, Vol. 16, N° 2, June 1984.

[Khoshafian 86] S. Khoshafian and G. Copeland, "Object Identity", Proc. OOPSLA 86, 1986.

[Kiernan87] Kiernan G., Morize I., "Building an Object-oriented Interface Over an Extended Relational Backend", Progress Report, ISIDE ESPRIT Project, N.221, Oct. 1987.

[Kuper 87] G. Kuper "Logic Programming with Sets", Proc. ACM PODS Conference, 1987.

[Maier 83] D. Maier and R. Lorie, "A Surrogate Concept for Engineering Databases", Proc. VLDB 9, 1983.

[Reiter84] Reiter R. : "Towards a Logical Reconstruction of Relational Database Theory", in On Conceptual Modelling, Book, Pp 191-234, Springer-Verlag Ed., 1984.

[Tsur 86] S. Tsur and C. Zaniolo, "LDL: A Logic Based Data Language", Proc. VLDB 12, 1986.

[Woelk 86] D. Woelk, W. Kim and W. Luther, "An Object-Oriented Approach for Multimedia Databases", Proc. ACM-SIGMOD 86, 1986.

[Zaniolo 85] C. Zaniolo, "The Representation and Deductive Retrieval of Complex Objects", Proc. VLDB 11, Stockolm, 1985.