A Logic-Programming/Object-Oriented Cocktail

François Bancilhon

MCC
9430 Research Blvd
Austin, Tx, 78759
USA

ABSTRACT

I define a simple data model which is based on an object oriented approach and uses logic programming as a computational model. The query part of the data model is purely declarative and has a fixpoint semantics, while the update part is imperative and uses assignments.

1. Introduction.

The object oriented approach has generated considerable interest in the database community. It seems to be particulary well fitted to handle new types of applications such as CAD, software and AI. However the natural extension to relational datbase technology is in fact the logic programming paradigm more than the object oriented one. The question I want to adress here is whether the two paradigms are compatible.

This work is based on the assumption that the choice of an object oriented approach is orthogonal to the choice of a specific computational model (imperative programming, logic programming or functional programming for instance). So, I briefly describe what I understand to be the characteristics of an object oriented approach, then I present a model which retains these characteristics while using logic as a computational model. Finally I discuss the notion of update, arguing that updates are by nature linked to the problem of assignment.

2. Object Oriented Characteristics.

I define the object oriented approach as the following set of concepts: (i) Encapsulation and information hiding (the objects contain the operations to deal with them and are only accessible through those, this is the classical abstract data type idea). (ii) Object typing, type hierarchy and method inheritance. (iii) Message passing and overloading.

Notice that I did not put object identity in there. This is for two reasons: first because I could not find a definition of the concept which was not in terms of implementation (surrogates or pointers), second because this seems to me to be in full contradiction with the notion of encapsulation: encapsulation means that an object exists only by its behavior, while object identity says that two objects can have the same behavior and be different [Ait-Kaci 85]. I know this is not the place to be controversial, however, through the "dirty update" approach, the surrogate lovers will eventually be satisfied.

Finally, I also understand that there is something called "complex objects" (these are essentially hierarchical structures with repeating groups) which people (especially data base people) like. This notion, at least to my understanding has nothing to do with object oriented programming and but I have the feeling that some confusion exists here.

3. Object types:

An *object type* consists of (i) a state type, (ii) a set of methods and (iii) a set of method implementations. The only visible part of the object is the set of methods.

A *state type* is a list of variable names, each variable having an associated object type (i.e. the variable will range over an object type domain).

A *method* is a predicate name and its arity.

A *method implementation* is a set of Horn clauses. Those Horn clauses contain only simple variables, constants (i.e. strings or integers), variable names from the state structure and the special constant "me" which refers to the object itself. Implementations are also such that the first slot of the predicate on the left hand side of the rule is always "me".

Finally there are some predefined types: with their associated methods (arithmetic for instance), that are not detailed here.

Here is an example of an object type:

```
tperson =

   state:
        name: tname;
        birth_date: tdate;
        father: tperson;
        mother: tperson;

   method:
        FIRST_NAME(X,Y);
        LAST_NAME(X,Y);
        AGE(X,Y);
        FATHER_NAME(X,Y);
        PARENT(X,Y);
        ANCESTOR(X,Y);

   implementation:
        FIRST_NAME(me,X) :- FN(name,X).
        LAST_NAME(me,X) :- LN(name,X).
        AGE(me,X) :- YEAR(birth_date,Y),YEAR(current_date,Z),X is Z-Y.
        FATHER_NAME(me,Y) :- FIRST_NAME(father,Y).
        PARENT(me,father).
        PARENT(me,mother).
        ANCESTOR(me,X) :- PARENT(me,X).
        ANCESTOR(me,X) :- PARENT(me,Y),ANCESTOR(Y,X).

tname =

   state:
        first_name:string;
        last_name:string;

   method:
        FN(X,Y);
        LN(X,Y);

   implementation:
        FN(me,first_name).
        LN(me,lastname).
```

tdate =

state:
year: integer;
month: integer;
day: integer;

method:
YEAR(X,Y);
MONTH(X,Y);
DAY(X,Y);

implementation:
YEAR(me,year).
MONTH(me,month).
DAY(me,day).

An *instance* of an object type is obtained by assigning values to the variable names of the object state. These values are objects instances of the specified type. The definition is grounded by the existence of the atomic type, which are interpreted directly. It is also assumed that "nil" (the undefined value) is part of every domain extension. An example of a date instance is as follows:

[year = 1948; month = 4; day = 10]

An instance of a person is obtained by assigning an instance of a name to name, and instance of a date to date and instances of person to father and mother.

4. A first simple semantics:

It is defined by transforming the set of object types and object instances into a set of Horn clauses, in the following way:
(1) Associate a surrogate with each object instance.
(2) For each object create a set of rules from the object type implementation method by replacing in each rule "me" by the surrogate of the object instance and each variable name by its value. In case any variable gets replaced by nil just delete the clause.

Example:

Object type	*Object instance*	*surrogate*

tdate	year = 1948;
	month = 4;
	day = 10;
		d1

tdate	year = 1913;
	month = 1;
	day = 17
		d2

tdate	year = 1912;
	month = 6;
	day = 27
		d3

tname	first_name = john; last_name = doe;	n1
tname	first_name = mary;	n2

```
                last_name = doe;

tname           first_name = peter;     n3
                last_name = doe;

tperson         name = n1;              p1
                birth_date = d1;
                father = p2;
                mother = p3;

tperson         name = n2;              p2
                birth_date = d2;
                father = nil;
                mother = nil;

tperson         name = n3;              p3
                birth_date = d3;
                father = nil;
                mother = nil;
```

This generates the following set of Horn Clauses:

```
FIRST_NAME(p1,X) :- FN(n1,X).
LAST_NAME(p1,X) :- LN(n1,X).
AGE(p1,X) :- YEAR(d1,Y),YEAR(current_date,Z),X is Z-Y.
FATHER_NAME(p1,Y) :- FIRST_NAME(p2,Y).
PARENT(p1,p2).
PARENT(p1,p3).
ANCESTOR(p1,X) :- PARENT(p1,X).
ANCESTOR(p1,X) :- PARENT(p1,Y),ANCESTOR(Y,X).

FIRST_NAME(p2,X) :- FN(n2,X).
LAST_NAME(p2,X) :- LN(n2,X).
AGE(p2,X) :- YEAR(d2,Y),YEAR(current_date,Z),X is Z-Y.
ANCESTOR(p2,X) :- PARENT(p2,X).
ANCESTOR(p2,X) :- PARENT(p2,Y),ANCESTOR(Y,X).

FIRST_NAME(p3,X) :- FN(n3,X).
LAST_NAME(p3,X) :- LN(n3,X).
AGE(p3,X) :- YEAR(d3,Y),YEAR(current_date,Z),X is Z-Y.
ANCESTOR(p3,X) :- PARENT(p3,X).
ANCESTOR(p3,X) :- PARENT(p3,Y),ANCESTOR(Y,X).

YEAR(d1,1984).
MONTH(d1,4).
DAY(d1,10).

YEAR(d2,1913).
MONTH(d2,1).
DAY(d2,17).

YEAR(d3,1912).
MONTH(d3,6).
DAY(d3,27).
```

```
FN(n1,john).
LN(n1,doe).

FN(n2,mary).
LN(n2,doe).

FN(n3,peter).
LN(n3,doe).
```

Note that this is not a suggestion for an implementation, but simply a way of defining the semantics of a system of object types and object instances. One of the problems of this interpretation (from a performance point of view) is that if there is an average number of NO objects instances and if each object type has an average of NR rules then the total number of rules generated in the Horn interpretation is NO*NR. This means for instance that the rule for computing the ancestor is going to be generated as many times as there are persons in the database.
This scheme can now be simplified in the following fashion, which will make it amenable to a possible implementation: Whenever a rule contains no local variables then it is instantiated only at the type level by replacing "me" by some variable name. This would yield the following set of clauses for the previous example:

```
ANCESTOR(Z,X) :- PARENT(Z,X).
ANCESTOR(Z,X) :- PARENT(Z,Y),ANCESTOR(Y,X).

FIRST_NAME(p1,X) :- FN(n1,X).
LAST_NAME(p1,X) :- LN(n1,X).
AGE(p1,X) :- YEAR(d1,Y),YEAR(current_date,Z),X is Z-Y.
FATHER_NAME(p1,Y) :- FIRST_NAME(p2,Y).
PARENT(p1,p2).
PARENT(p1,p3).

FIRST_NAME(p2,X) :- FN(n2,X).
LAST_NAME(p2,X) :- LN(n2,X).
AGE(p2,X) :- YEAR(d2,Y),YEAR(current_date,Z),X is Z-Y.

FIRST_NAME(p3,X) :- FN(n3,X).
LAST_NAME(p3,X) :- LN(n3,X).
AGE(p3,X) :- YEAR(d3,Y),YEAR(current_date,Z),X is Z-Y.

YEAR(d1,1984).
MONTH(d1,4).
DAY(d1,10).

YEAR(d2,1913).
MONTH(d2,1).
DAY(d2,17).

YEAR(d3,1912).
MONTH(d3,6).
DAY(d3,27).

FN(n1,john).
LN(n1,doe).

FN(n2,mary).
```

LN(n2,doe).

FN(n3,peter).
LN(n3,doe).

This has the advantage of "abstracting" the definition of the ancestor relation at the type level.

5. A Message sending and receiving semantics.

Let us now turn to a more object oriented definition of the semantics. Each predicate P(x1,x2,...,xn) is seen as a message. It is essentially sent to every object in the database. Every object O which receives it does the following:

(1) x1 has to unify with object O

(2) P(x1,x2,...,xn) has to unify with at least one left hand side of one of the methods of O.

(3) If it does, unify P(x1,x2,...,xn) with every left hand side of methods in O. In this process some of the uninstantiated variables of x1,x2,...,xn will become instantiated.

(4) The reply to the message is a set of tuples (possibly empty if there is no match)

For instance FATHER(p1,X) will send message FATHER to object p1 and p1 will instantiate X to some object, FATHER(X,Y) will send message FATHER to all objects in the database and those having a FATHER method will instantiate X to themselves and Y to whatever their method says, and FATHER(X,p1) will send the message FATHER with the second argument bound to p1 to every object in the data base and only those having FATHER as a method and being able to unify with this message will reply by instantiating X to themselves.

The next step is to describe how a message is evaluated when received: when object O receives message P(x1,x2,...,xn) it first unifies x1 with himself, then for every clause whose left hand side unifies with P(x1,x2,...,xn) it performs the unification, instantiates the variables of the right hand side according to that substitution and send these messages accordingly... So, this is just the standard operational semantics of logic programming.

6. A fixpoint semantics:

Let us define formally the answer of an object to a message:
(1) If o owns the method P(o,X) :- P1(o1,X1,X2,...,Xn)
If oi replies (oi,xi) to Pi(oi,xi) (for i= 1,2,...,n)
If Pi(oi,yi) matches Pi(Oi,Xi) under substitution σ,(for i= 1,2,...,n)
Then o replies (o,σ(X)) to P(o,σ(X)).
(2) If o owns the method P(o,x) then it replies (o,x) to P(o,x).
(3) If o replies (o,x) to (o,X) and if Y $<$ X (in the sense that there is valuation of some of the variables in X which generates Y) then o replies (o,x) to (o,Y).

Taking as a partial order set inclusion for the answer set to a message, one can define the fixpoint semantics of a message answer (Yes, I do agree this part is sketchy).

7. An Alternative Model

I now address the problem of "complex objects". This will be done by having the state of the object represented by a hierarchical record with repeating groups (COBOL fans, here you are). As before, an object type consists of (i) a state type, (ii) a set of methods and (iii) a set of

method implementations.

Intuitively a state is represented by a hierarchical structure defined from objects using the tuple and set construct.
More formally, the state is a structure built as follows:
(i) Elementary structures: An object type is a structure.
(ii)Tuple construct: If t1,t2,...,tn are structure and n1,n2,...,nn are names then [n1:t1,n2:t2,...,nn:tn] is a structure.
(iii) Set construct: If t is a structure and n is a name then {n:t} is a structure.

As before, a method is a predicate name and its arity.

A method implementation is a set of Horn clauses; those Horn clauses contain only variables, constants (i.e. strings or integers), structure variables from the state structure and the special constant "me" which refers to the object itself. The only difference with the previous model is that state variables are replaced by structure variables: a *structure variable* represent an object from the structure state. It is therefore a sequence of labels which lead to a leaf of the structure. Labels are separated by "." in the sequence. Thus *parent.father* references the *father* field of the *parent* tuple and *children.child* designates a typical *child* from the *children* set. As before, implementations are such that the first slot of the predicate on the left hand side of the rule is always "me".

Here is an example of an object type:

```
tperson =

   state:
        [name: tname;
         birth_date: [day: integer; month: integer; year: integer];
         parent: [father: tperson,
                  mother: tperson]
         children: {child: tperson}]

   method:
        FIRST_NAME(X,Y);
        LAST_NAME(X,Y);
        AGE(X,Y);
        FATHER_NAME(X,Y);
        PARENT(X,Y);
        DESCENDANT(X,Y);

   implementation:
        FIRST_NAME(me,X) :- FN(name,X).
        LAST_NAME(me,X) :- LN(name,X).
        AGE(me,X) :- X is birth_date.year - current_year
        FATHER_NAME(me,Y) :- FIRST_NAME(parent.father,Y).
        PARENT(me,parent.father).
        PARENT(me,parent.mother).
        DESCENDANT(me,children.child).
        DESCENDANT(me,X) :- DESCENDANT(children.child,X).

tname =

   state:
        [first_name:string;
         last_name:string;]
```

```
method:
    FN(X,Y);
    LN(X,Y);

implementation:
    FN(me,first_name).
    LN(me,lastname).
```

An instance of an object type is obtained by assigning a values to the state structure:

Object type	*Object instance*		*Surrogate*
tname	first_name = john;		n1
	last_name = doe;		
tname	first_name = mary;		n2
	last_name = doe;		
tname	first_name = peter;		n3
	last_name = doe;		
tperson	name = n1;		p1
	birth_date =	[year = 1948;	
		month = 4;	
		day = 10;]	
	parent =	[father = p2;	
		mother = p3]	
	children = {}		
tperson	name = n2;		p2
	birth_date =	[year = 1913;	
		month = 1;	
		day = 17]	
	parent =	[father = nil;	
		mother = nil]	
	children = { p1 }		
tperson	name = t3;		p3
	birth_date =	[year = 1912;	
		month = 6;	
		day = 27]	
	parent =	[father = nil;	
		mother = nil]	
	children = { p1 }		

So this is just a trick to introduce hierarchical records. The only thing which is a little difficult to define is the use of the set construct (since the tuple construct can simply be seen as a notational convenience to structure a set of values). The convention adopted here is that when a message is sent to the representative element of a set (child for instance) it is sent to all the elements of the set in question (children in this case). Up to this convention, the semantics of this extended model is a straightforward extension of the semantics of the original model.

8. Type hierarchy and method inheritance.

The type hierarchy is strict and explicitly declared by the user. Therefore "t1 subtype_of t2" means that t1 inherits (i) the set of methods of t2 and (ii) the state variables of t2. Of course I can add any new method or state variables to t1, which specializes the type. The convention is that only the new variables and methods are redeclared. I do not address the problem of multiple inheritance or of method overriding, which are outside of the scope of this paper. Here is an example of type specialization:

```
tperson =

    state:
        name:tname;
        age:integer;

    methods:
        FIRST_NAME(X,Y);
        LAST_NAME(X,Y);
        OLDER(X);

temployee = subtype of tperson;

    state:
        ss#:integer;
        salary:integer;

    methods:
        INCREASE_SALARY(X,Y);
        SEARCH(X,Y);
```

In case the state is a complex structure, hierarchy will have to require a more complex relationship between the two state structures (essentially the parent type must be a subtree of the child type with same root).

9. Updates.

Up to now, I haven't talked about any updates, i.e. I have assumed that somebody was magically creating the set of objects in the beginning and that the only thing I did afterwards was query it. My feeling is that it is a not such a bad way of looking at the problem. Looking at the existing systems, one can see that (i) the notion of an update is a natural one (i.e. people want to modify data) (ii) every knowledge representation scheme which has clean semantics does not have clean semantics of updates (e.g. PROLOG with "assert" and "retract" or the relational model with "insert" and "delete") and the logic and database school has failed to provide us with any update semantics. (iii) On the contrary, imperative programming languages, from BASIC to SMALLTALK have emphasized update through the concept of assignment but they do not have clean fixpoint semantics.

I suggest here to admit this dichotomy (like PROLOG and relational system do) and to have clean queries defined the way suggested above and updates specified using assignments and imperative programming. At this point I don't care what imperative language is chosen (why not PASCAL?). So the rule is that I'll write clean queries using only the logic programming approach, and I'll write dirty updates using both imperative and logic programming (I might still have a few problems with that later mixture but I can probably fix it in some ad hoc fashion)

An example of such a use of "Dirty Updates" could be:

```
tperson =
```

```
   state:
         name: tname;
         age: integer;

   method:
         OLDER(X);
         CHANGE_NAME(X,Y);

   implementation:
         OLDER(X) :- (age := age + 1).
         CHANGE_NAME(X,Y) :- CFN(X,Y).

tname =

   state:
        first_name:string;
        last_name:string;

   method:
          CFN(X,Y);
          CLN(X,Y);

   implementation:
          CFN(X,Y) :- (first_name := Y).
          CLN(X,Y) :- (last_name := Y).
```

10. Conclusion

A number of details must be ironed out to make this proposal complete: we have adopted the idea that Horn clauses should only contain simple variables, thus forbidding function symbols in the terms. One justification for that choice is that the representation of complex objects is done by structured states and it is possible that function symbols won't be needed any more, but this issue has to be further investigated.
The decision to have the first attribute of each predicate used to represent the object itself is not very natural and is assymetric. Better solutions might be discussed.
Finally the imperative and logic mixture for the update part has to find an acceptable solution.

It is now worth testing with both schools of thought whether this cocktail matches their taste.

11. Acknowledgements

The idea presented here was generated with David Dewitt.

12. References

[Afsarmanesh 85]
"An Extensible Object-Oriented Approach to Databases for VLSI/CAD" H. Afsarmanesh, D. Knapp, D. McLeod and A. Parker Tech Report 85-330, Computer Science Dept, UCLA

[Ait-Kaci 84]
"A Lattice Theoretic Approach to Computation Based on a Calculus of Partially Ordered Type Structures" H. Ait-Kaci PhD, University of Pennsylvania, 1984.

[Ait-Kaci 85]
Hassan Ait-Kaci, "Unpublished Gripe", March 1985.

[Apt 82]
"Contribution to the Theory of Logic Programming" K. Apt and M. Van Emden JACM, Vol 29, No 3, July 1982, pp 843-862

[Cardelli 84]
"A Semantics of Multiple Inheritance" L. Cardelli in "Semantics of Data Types" Lecture Notes in Computer Science, No 173, pp 51-67, Springer Verlag 1984.

[Chikayama 84]
"Unique Features of ESP" T. Chikayama Proceedings of the International Conference on Fifth Generation Computer Systems, 1984

[Copeland 84]
"Making Smalltalk a Database System" G. Copeland and D. Maier Proceeding 84 SIGMOD Conference, june 1984. Data Models, Object Oriented Data Model, Time.

[Ishikawa 84]
"The Design of an Object Oriented Architecture" Y. Ishikawa and M. Tokoro

[Furukawa 84]
"Mandala: A Logic Based Knowledge Programming System" K. Furukawa, A. Takeushi, S. Kunifuji, H. Yasukawa, M. Ohki, K. Ueda Proceedings of the International Conference on Fifth Generation Computer Systems, 1984

[Mizoguchi 84]
"LOOKS: Knowledge Representation System for Designing Expert Systems in a Logic Programming Framework" F. Mizoguchi, H. Ohwada and Y. Katayama Proceedings of the International Conference on Fifth Generation Computer Systems, 1984

[Nakashima 83]
"Data Abstraction in Prolog/KR" H. Nakashima and N. Suzuki New Generation Computing, 1 (1983) pp 49-62

[Shapiro 83]
"Object Oriented Programming in Concurrent Prolog" E. Shapiro and A. Takeushi New Generation Computing, 1 (1983) 25-48 Object Oriented programming and Prolog.

[Tokoro 84]
"An Object Oriented Approach to Knowledge Systems" M. Tokoro and Y. Ishikawa Proceedings of the International Conference on Fifth Generation Computer Systems, 1984 Logic and Object Oriented Programming.

[Zaniolo 85]
"Object Oriented database Systems and Knowledge Systems" C. Zaniolo. H. Ait-Kaci, D. Beech, S.Cammarata, L. Kerschberg and D. Maier Report from the Kiowa Working Group