Merging Objects and Logic Programming: Relational Semantics

Herve Gallaire

European Computer-Industry Research Centre (E.C.R.C)
Arabellastr. 17, D-8000 Muenchen 81 FRG

Abstract

This paper proposes new semantics for merging object programming into logic programming. It differs from previous attempts in that it takes a relational view of method evaluation and inheritance mechanisms originating from object programming. A tight integration is presented, an extended rationale for adopting a success/failure semantics of backtrackable methods calls and for authorizing variable object calls is given. New method types dealing with non monotonicity and determinism necessary for this tight integration are discussed. The need for higher functions is justified from a user point of view, as well as from an implementation one. The system POL is only a piece of a more ambitious goal which is to merge logic programming, object programming and semantic data models which can be seen as an attempt to bridge the gap between AI and databases. The paper is restricted to a programming perspective.

1. INTRODUCTION

This paper is not about yet another mix of logic programming and object programming. Its goals are at a different level than those of most systems which have been presented up to now [1],[2],[3],[4],[5] which merely copy object- programming semantics [6],[7],[8] into a logical one. The question addressed here is whether the notions of inheritance and of procedural semantics attached to the object programming systems can be kept, or whether they have to be revisited in the light of a relational rather than functional paradigm. The answer given in this paper is that these concepts should indeed be defined differently in this context, and a complete solution is presented. This requires to adopt a success/failure semantics of backtrackable method calls, instead of the call/return classic mechanism. It also requires to allow for variable object calls and to introduce new types of methods dealing with non monotonicity and determinism. A for these decisions is given.

Methods and method calling are discussed extensively here, but slots are not because they are orthogonal to the actual topics of interest in this paper. Similarly the paper does not analyse objects in the perspective of parallel execution as object programming and logic are still very much sequential. The problems studied do not suffer from these limits.

Another important thrust of this study has been to develop ideas leading to realistic implementations of the complex operations required in a full relational context as proposed in this paper, but they are not discussed here.

Early applications of this system have shown how this integration could still be improved. Several more complex operators have been defined to give adequate tools to application developers in order to integrate operations to method calls (eg average,..) while retaining the efficiency of the method evaluation technique developed. In many ways they are the corresponding operators of the classical 'set-of' one in a logic-oriented framework, and in some sense of the aggregation operators added to the relational algebra in a database framework.

The rest of the paper is divided into four further sections and a conclusion. Section 2 states the requirements set up to build the POL system. Section 3 defines the syntax of POL. Section 4 deals with the semantics, especially the method call interpretation. Section 5 presents the higher order operators. Initial ideas about the POL system have been presented briefly in [9].

The work described here is part of a wider effort to bridge the gap between knowledge representation techniques as used in the A.I (objects, frames) and in the Database communities (entities, relationships.....), based on the use of logic as a possible unifying framework. Obviously POL bears relations to languages developed from the other ends: AI languages extended to handle inference [10], [11]. Theorem Provers using theory resolution [17], semantic database systems [12] offering inference mechanisms [13], [14], [15]. It is anticipated that there will be a strong convergence through approaches of this type between the programming, the AI and the database fields.

POL has been built through progressive refinement, introducing first an interpreter of method calls; then a simple compiled version has been developed and progressive optimisations which appeared necessary to fulfil the additional requirements were introduced later.

2. REQUIREMENTS

The following sums up the main ones retained for the development of POL.

<u>req1</u> : the POL system must be a superset of Prolog, any Prolog program should run unchanged.

<u>req2</u> : the POL system must allow for a programming based on objects, classes and method calls. Inheritance should be a feature of POL, including multiple inheritance. Slots are not discussed here,but they are supported.

<u>req3</u> : the relational framework must be used to refine appropriately concepts such as inheritance and method evaluation, as the usual function-based semantics is not appropriate

req4 : the main features of logic programming, namely the logic variable, non-determinism (including backtracking to implement it) should be used throughout. Thus method calls must be fully general and backtrackable. They provide fully associative retrieval of sets of objects.

Requirements 1 and 2 are straightforward. Requirements 3 and 4 give preeminence to logic. The semantics of object programming is for the most part system dependent and this paper adds to the long list of such contributions with specific motivations.

req5 : the system must allow dynamic creation and deletion of objects, classes, methods.

There are additional requirements which tie the system to the semantic models domain. Although they are out of the scope of this discussion, some of the features supported by POL are relationships [12] and fully deductive relationships.

3. SYNTAX

A sentence in POL is a clause belonging to Prolog+ or a declaration:

Prolog+. A clause in Prolog+ is a Prolog clause, including additional built-in evaluable predicates (defined for most of them as Prolog operators) ; in particular the infix operator ':' to indicate a method call. Method calls are written '**X:Y**' where X is an object (instance), or a Prolog variable, and where Y is a Prolog literal which must have been defined in an associated method declaration (see below); the parameters of Y can be objects or variables indifferently. Other operators introduced to deal with higher order functions are not discussed here. As usual in Prolog, upper case letters denote variables, while lower case letters denote constants. The added built-in predicates are simulated in the current implementation, but they are to be understood as true built-in, in a final implementation.

Declarations of objects (instances) and classes.There are two predicates: '**X isa Y**' for class hierarchy and '**X instance Y**' for class instances. Multiple hierarchies are possible.

Declarations of methods. Their syntax is '**X with Y**' or '**X withdefault Y**' or '**X withdeterministic Y**', where **X** is a class and **Y** is a Prolog clause 'Call:-Body'. Methods have types given by their declarations. They correspond to classic, default or deterministic methods whose role is explained later in section 4. The 'Call' part of the declaration is a method call (as described above) 'U:V' where 'U' must be a variable which will denote at call time the object on which the method is called or a true variable when doing object retrieval through method calling. 'Body' is the body of the method to be executed when the method is called. It is any clause of Prolog+ defined above, ie it can itself include calls to methods, using if needed 'U' to work on the same object. Note that additionally, 'U' may denote the class itself at which the method is defined, ie the method can be associated with the class and not with the objects of the class. See the semantics for a justification; use of this to model complex domains has been demonstrated in various applications. Thus the syntax given to users is Prolog to which few syntactic frills have been added.

4. SEMANTICS

There are four basic notions coming from the object world, namely class, object, method, inheritance. Here there is but one class concept, hence no separate metaclass concept is used; objects are separate from classes. Methods are defined, attached to a class and operate on instances (direct or indirect) of that class. Moreover, if so desired methods can operate on classes, to attach information to them rather than to their instances obtaining metaclass features. As example, the generic structure of a type of documents is valid for that type, not for the instances of that type which have another concrete structure possibly derived from the generic one: thus the generic structure would be attached to the class itself. POL requires objects to have names. We will note method calls
X:methodname(Parameter), or
X:methodname(Parameter1, Parameter2). It is perhaps easier to view this notation as an alternative for another predicate, methodname(X,Parameter) or
methodname(X,Parameter**1**,Parameter**2**), etc.

4.1. Basic Choices

The first point of interest to discuss has to do with the evaluation of a method call. In classical object programming system, a call Object:methodname(Parameter) is
usually answered according to the following rules :
take the first method according to the inheritance rules which has the name of the method call, evaluate that method for the couple (Object,Parameter), one of which is (almost) always a constant (Object), the other a variable (Parameter); only one such couple will be used; the call may then instantiate or not Parameter - usually it would do so. In this respect, the method call behaves exactly as a procedure call with a call/return paradigm. Of course some languages, e.g flavors in [6] offer ways to combine the values of relevant methods, but this is still in the functional context.

In the logic framework both aspects of the above rules must be questioned. First even if logic provides also a procedural interpretation, its main interest stems from a declarative interpretation which corresponds to a success/failure paradigm rather than to a call/return one. Thus it is appealling to modify and adapt the method call to the success/failure paradigm. This change has consequences that are analysed later. Similarly, but this is obvious, the constraint of 'one' answer must be released to fit the relational environment. Thus each method call will be backtrackable.
To clarify the issues we summarise the various possibilities :

Case a : a call **object:methodname(Parameter)**, where object is a known object, not a variable. This is called a **constant object call**. There are three possible independent semantic interpretation choices:
a1 - call/return versus success/failure paradigm which influences the notion of "first answer"
a2 - multiple answers or single answer of the selected method by a1
a3 - multiple methods calls or single method call to provide additional answers when possible
Most systems choose call/return, single answer to a2, and single method to a3. Instead POL implements success/failure, multiple answers to a2, multiple methods to a3. Obviously the programmer will have the possibility to control this and to accept only one method, one answer, etc. This also corresponds to ESP [1], but see further comments. In [3] are provided a success/failure paradigm, multiple answers, single method.

Case b : a call X:method(Parameter). where X is a free variable in the logic sense. Then the question is whether such a call. referred to as variable object call in the sequel. is correctly and efficiently handled, in all contexts. i.e. even in contexts where non-monotonicity is handled. In [3] this is not dealt with: instead. as in many other systems, one has to program this search at an external level. Consider a predicate 'findallobjects' defined as:

findallobjects(Nickname):- Obj instance Something,
Obj:whatis(Nickname)

where 'whatis' could be a user-defined method returning the nickname of the object it is applied to. The call mechanism involving the complex evaluation scheme of ':' will be reactivated for each object, a penalty. It is not possible to only have a pure loop around the method evaluation in order to produce object variable calls. This would also have the obvious serious problem of redundant answers in lattices. Another problem due to the way defaults (i.e. non-monotonicity) are implemented in [3], would be the loss of completeness in such calls, ie not all answers would be obtained. Similar remarks apply to ESP [1]. One system seems to address such calls, Sidur [2], but the programmer has to give a 'program' to do so. POL does solve these problems completely, but requires a much more complex implementation to be efficient, and new types of methods.

Here are the rules retained; justifications are given afterwards:

(R1) a method is deemed to provide an answer to a method call when and only when the method predicate evaluates to true. Thus when a method predicate evaluates to false, the search for an answer goes on according to inheritance rules. This is the success/failure paradigm and it allows to simulate the call/return paradigm.

(R2) inheritance is basically from bottom-up in the hierarchy of classes (which imposes to write the declaration of the methods with identical names corresponding to the lowest classes first). At a given level, siblings are examined in the order of declarations of the methods (not of the hierarchy).

The above comparisons dealt with other systems integrating objects and logic programming. It must be noted that at the opposite end of the spectrum, a language like Gemstone [16] which unifies Smalltalk and databases obviously offers variable objects thanks to its set notation. However it does not offer any deductive capabilities. While we are reviewing hybrid systems, let us again mention [10],[11], which combine various modelling aspects and deductive features, at least to some extent. They do not appear to offer the variable object feature.

4.2. Success/failure versus call/return paradigms

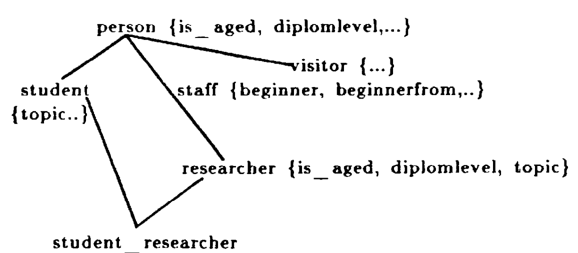

Figure1 describes a hierarchy of classes, with method-names. Corresponding to it we would have the following:

```
....
student_researcher isa researcher                         (d1)
student_researcher isa student                            (d2)
researcher isa staff                                      (d3)
staff isa person                                          (d4)
...
pat instance researcher                                   (d7)
ida instance researcher                                   (d8)
franz instance student_researcher                         (d9)
age(ida,25)                                               (f1)
age(pat,35)                                               (f2)
age(franz,no_value)                                       (f3)
...
researcher with X:is_aged(Y):-age(X,Y),Y=/=no_value       (r1)
researcher with X:diplomlevel(Y):-dip(X,Y),Y=/=no_value   (r2)
staff with X:beginner:- X:diplomlevel(Y),Y=<4,            (r3)
         X:experiencelevel(Z), Z=<3
......
person with X:is_aged(Y):-askuser(age,X,Y)                (r5)
person with X:diplomlevel(Y):-askuser(diplomlevel,X,Y)    (r6)
researcher with X:topic(Y):-X:in_team(Z),topics(Z,Y)      (r7)
student with X:topic(Y):-record(X,Y)                      (r8)
```

Examples in this paragraph will be taken from Figure1 which describes a rather simple domain. They will be used to justify the success/failure paradigm as opposed to the call/return one. This choice does not prevent from simulating the call/return evaluation as it would be a simple matter to introduce a control such as a '!' (cut) at the beginning of each of method body. As will be seen however, more elaborate solutions will have to be found to properly take into account the variable object calls. Assume a query such as

?-franz:is_aged(Y)

Such a query in an object programming system with call/return features. would yield as answer a free variable as it would be deemed to have had an answer given at class researcher where the 'procedure' is_aged is defined (but fails in the logic programming sense), hence yielding the free variable answer. or perhaps an error. To palliate this. a 'sendsuper' call. classical in object systems, could be used in the body of is_aged. but this is highly procedural in nature and thus should probably not be used in this context. It would present other problems as well (see next). Another more problematic example would indeed be

?-franz:topic(Z)

As topic is defined first at class researcher before being defined at class student. if one is to take the call/return paradigm, it will evaluate that and only that method. This will call the procedures in the body of 'topic' defined at class researcher and return again a free variable for Z if franz is not registered in any team (he may instead have a topic which is his university topic, not given by the company). This means that the method call in researcher class fails and thus the search stops. On the contrary, the success/fail paradigm would by itself continue the search correctly. Here a sendsuper mechanism would not work as the search has to proceed in a different branch of the inheritance graph. Of course adopting the success/failure paradigm does not mean that one is necessarily interested in all answers and there should be ways to control that search. We turn to that now.

4.3. Default Methods

Let us now assume the query

?-pat:is_aged(Y)

with $S/\overline{F}$ (success/failure) as the call mechanism semantics. This would yield a first answer "35" using data of Figure1 and when backtracked over, would call for 'askuser', an obviously non-desirable feature. Prolog programmers would thus rewrite **(r1)** as

researcher with X:is_aged(Y):- age(X,Y),Y=/=no-value,! **(r1a)**

which solves this problem, by stopping propagation. But then, what about the query
?- **Who:is_aged(Y)**
i.e. what about variable object calls for such a query? The '!' which is in (**r1a**) prevents any backtrack. i.e. only one answer is obtained. There is no way to combine both features. The solution adopted in POL is to introduce a different type of method, so-called 'default' methods. Doing so, having a clear default semantics will put the burden on the structuring of methods and objects, not on coding individual bodies of methods. One would thus keep (**r1**) and further replace (**r5**) by (**r5a**) :

person withdefault X:is_aged(Y):- askuser(age,X,Y) (**r5a**)

Additionally (**r1**) could be replaced by (**r1a**), but this would just be for (small) efficiency gains.
Thus the semantics of ':' evaluation and of default must be defined as follows:
(**R3**) a default method is used on object X (constant) if and only if no earlier call of the same method has succeeded in the current call evaluation. Here, earlier is to be understood according to **R2** in 4.1.
(**R4**) variable object calls must work on all objects of relevant classes, no matter the contents (bodies) of the methods (i.e. even if those contain cuts). **R4** is the one difficult rule to implement correctly and efficiently.

These rules form the basis for the associative retrieval functions. The presence of such calls with variable objects has another important consequence that is now to be reviewed.

4.4. Deterministic Methods

Granted that we make provision for variable object calls, implying backtracking to get all relevant objects, when there are several (inherited) methods (other than default ones) applying to a given class, each object will be applied to all methods relevant for that class. This may be redundant in some cases or desired; thus there must be ways to control that process. For instance assuming that 'topic' was also defined at class staff in addition to being defined at class researcher, it is likely that one would want to evaluate only one 'topic' method for a given researcher:

...
researcher with X:topic(Y):- body1 (**r7**)
student with X:topic(Y):- body3 (**r8**)
staff with X:topic(Y):- body2 (**r9**)

It is possible to control the use of methods in different classes by using the universal and versatile '!' (cut) of Prolog once again, e.g. by replacing (**r7**) by

researcher with X:topic(Y):- body1,! (**r7a**)

This '!' does not cause problems as far as evaluation of calls such as X:topic(Y) thanks to the implementation of ':' discussed at the end of subsection 4.3. This can be contrasted to [1],[3] - see the discussion in the introduction of section 4. But consider its semantics : it would in fact cut paths for branches which use true multiple inheritance. The few systems which have addressed this problem normally considered there was an OR between all methods, thus any cut had to stop all evaluations. It is more flexible to consider that any cut is to be only local to a branch in which it appears. Thus any cut in body3 in (**r8**) will not affect (**r9**) use. On the other hand it is sometimes necessary to have such methods which do stop evaluation on all branches : to do so we have introduced another type of method, deterministic ones.

(**R5**) : When it anwers (i.e succeeds) a <u>deterministic method</u> stops all calls for the same object when backtrack occurs, including on sibling methods.

Thus such methods correspond to high priority ones. Assume that in the example we want to give stronger preference to topics defined in the research centre than to topics defined in relation to external bodies. then we would replace (**r7**), (**r9**) with (**r7a**) as above and (**r9a**) :

staff withdeterministic X:topic(Y):- body2 (**r9a**)

Here are a few possible methods calls and their possible answers :

?-ida:topic(Y)	Y=knowledge-bases could be given by (**r7a**) -no more answer can be obtained. ida is a researcher and the path to (**r9a**) is cut in (**r7a**); as ida is not a student at the same time no link to (**r8**) exists
?-franz:topic(Y)	Y=decision-making could be given by (**r9a**) assuming (**r7a**) fails for franz; even though franz is also a student, (**r8**) will not be tried as (**r9a**) is deterministic
?-joe:topic(Y)	assuming joe is a student-researcher, and assuming that (**r7a**) and (**r9a**) fail for him, then (**r8**) would be tried and might yield compilation-techniques as answer. No more search is involved
?-john:topic(Y)	Y=knowledge-bases and Y=logic, could be given by (**r7a**) and (**r8**), assuming john is a student-researcher whose work on knowledge bases is given by the non-deterministic rule (**r7a**), cutting the path to the deterministic (**r9a**) but leaving open the path to (**r8**)

and of course, thanks to the ':' evaluation (**R4**)
?-X:topic(Y) would then yield (assuming the above data) :
X = ida, Y = knowledge bases
X = franz, Y = decision-making
X = joe, Y = compilation-techniques
X = john, Y = knowledge bases
X = john, Y = logic

but nothing for X = pat in case none of (**r7a**), (**r8**) (**r9a**) succeed for him.

What has just been discussed is, of course, a way to have non-monotonic behaviour. While such a behaviour is easily obtained in, e.g. [1], [3], where variable object calls are not dealt with, it has to be specifically produced in this more complex and complete context.

4.5. Summary

Rules (**R1**) through (**R5**) give the semantics of the method call mechanisms. They differ significantly from traditional ways of evaluating method calls but can be reduced to them by the programmer if he/she wishes to do so. Completeness of evaluation relative to these rules requires careful implementation of variable object calls.

5. SOME HIGHER ORDER FUNCTIONS

Following the running example, let us imagine the query is to find all beginners from the class staff only, i.e. not all of them. The obvious query to do that is ' ?-X:beginner, X instance staff', unless it is '?-X instance staff, X:beginner'.

Both queries have significant drawbacks. The problem of where to put the generator or the test (in the first query, X instance staff is a test, a filter, while in the second query it is a generator) is classic in the database field, and not well solved. The first query generates (with the high cost of method evaluation) too many values for X, the second generates people (probably too many) and loses any optimisations of the call X:beginner which have been discussed in section 4 for variable object calls because X is now a constant which has just been generated. What is needed here is to have a restriction operator for ':', namely the built-in evaluable predicate 'inclasses(X,Listofclasses,X:method(Y))' which provides a kind of many-sorted logic in its implementation. This operator has been implemented and yields significant improvements in performance, when used. But it is only one of a few others which have proved quite useful; a discussion of these is out of scope here, and only examples of their use are given, continuing the running example. Before doing so, let us stress that all these higher level operators share the goal to restrict the search as intimately as possible in the ':' evaluation process. This is why they are useful. They are very similar in spirit to the classical set-of operator of Prolog, but they work in the more complex context of classes and method calls. All following predicates are also built-in evaluable ones.

Examples :

laocwc ([researcher], X:has_children(Y),L)
lists couples(an instance of researcher,a child)

laocwcb ([researcher], X:has_children(Y),Y,L)
lists instances of researcher for whom X:has_children(Y) succeeds; Y is hidden

laocwcbis ([researcher],X,Y^(X:has_children(Y)),L) differs from the previous one because the constraint could have been any predicate, not just a call to a method; cannot be used in all contexts

dfaoc ([staff], X:is_aged(Y), tally(Y,Z), avge(Z,Av))
computes the average-age of a set of instances of class staff.

Obviously such unaesthetic operators can be hidden, and one could define :

averageage (Listofclasses,Averageage) or
average(Something,Listofclasses,Average)

Using these operators not only simplifies considerably applications writing but also helps in getting satisfactory performance.

Finally, there is another available set of evaluable operators which allows the schema to be queried, i.e. the set of declarations itself so as to give metalevel features : these operators are either global to the schema (e.g. list all methods, etc.) or local to a class (list all methods attached to it,....) or to a method/relationship. Manipulating such a schema gives a way for end-users to understand what is described and manipulated in the knowledge base.

CONCLUSION The ideas presented here appear to be original in that they try to really merge several formalisms : logic programming, object programming (and semantic data modelling), rather than purely juxtaposing them in a common formalism as seems to have been the main thrust of the work on merging objects and logic up to now. The system POL can be characterized by the fact that it offers a full treatment of variable object calls, and of default as well as deterministic mechanisms. Its implementation has been designed to deal with the above in the non-obvious way. Finally it proposes higher level operators which are original and have a high pay-off both in efficiency and for application development. A full system should be developed to go beyond the experimental stage described here, but the basics of the implementation need only be carried over. It is hoped that ideas along these lines can contribute to the emergence of a better breed of logic languages as it is obvious that the need of systems offering semantic modelling capabilities jointly with inference capabilities is not yet satisfied- let alone if we add database requirements.

Acknowledgments This work has benefitted from discussions with the Logic Programming group of ECRC, especially David Chan and Reinhard Enders.

REFERENCES

[1] Chikayama, T. : Unique features of ESP. Proceedings FGCS'84, ICOT, Tokyo, November 1984, pp. 292-298

[2] Kogan, D., Freiling, M. : SIDUR - A structuring formalism for knowledge information processing systems. Proceedings FGCS'84, ICOT, Tokyo, November 1984, pp. 596-605

[3] Zaniolo, C. : Object-oriented programming in Prolog. Proceedings 1984 International Symposium on Logic Programming, Atlantic City, February 1984, pp. 265-270

[4] Enders, R., Chan, D. : Specification of BOP - an object-oriented extension to Prolog. ECRC, Technical Report LP-2, March 1985

[5] Furukawa, K., et al. : Mandala, a logic based knowledge programming system. Proceedings FGCS'84, ICOT, Tokyo, November 1984, pp. 613-622

[6] Moon, D., Stallman, R., Weinreb, D. : Lisp Machine manual 5th ed., MIT AI Laboratory 1983

[7] Goldberg, A., Robson, D. : Smalltalk-80. The language and its implementation, Addison-Wesley, 1983

[8] Bobrow, D., Stefik, M. : The Loops manual. Xerox PARC, 1983

[9] Gallaire, H. : Logic Programming - further developments. Proceedings of IEEE Symposium on Logic Programming , Boston, July 1985

[10] Brachman, R., Pigman Gilbert, V., Levesque, H.J. : An essential hybrid reasoning system - knowledge and symbol level accounts of KRYPTON. Proceedings IJCAI-85, pp. 532-539

[11] Vilain, M. : the restricted language architecture of a hybrid representation system. Proceedings IJCAI-85, pp. 547-551

[12] Chen, P. : The entity-relationship approach: towards a unified view of data. ACM Transactions on Database Systems, Vol. 1, No. 1, March 1976, pp. 9-36

[13] Stonebraker, M. : Thoughts on new and extended data models. Journees ADI - Bases de donnees avancees, Saint Pierre de Chartreuse, March 1985

[14] Gallaire, H., Minker, J., Nicolas, J.M. : Logic and databases, a deductive approach. Computing Surveys, Vol. 16, No. 2, June 1984, pp. 153-185

[15] Lloyd, J., Topor, R.W. : A basis for deductive database systems. TR 85-1, University of Melbourne, revised version, April 1985

[16] Copeland, G., Maier, D. : Making Smalltalk a database system. ACM 0-89791-128-8/84/006/0316, pp. 316-325

[17] Stickel, M.E. : Automated deduction by theory resolution. Journal of Automated Reasoning 1, 1985, pp333-355