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. Minimalism necessary for this tight integration are discussed. 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 larger effort to bridge the gap between knowledge representation techniques as used in the A.I. (objects, frames) and in the Database community (entities, relationships...), based on the use of logic as a possible unifying framework. Obviously POL bears relations to languages developed from the other ends: A.I. languages extending 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 fulfill the additional requirements were introduced later.

2. REQUIREMENTS

The following sums up the main ones retained for the development of POL.
req1: the POL system must be a superset of Prolog, any Prolog program should run unchanged.
req2: 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.
req3: 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.
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 declaration: 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) 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 without Y' or 'X with deterministic 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, 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, in 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. 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, Parameter1, Parameter2), 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 ISi 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 appealing 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 backtracking.

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.
4.2. Success/failure versus call/return paradigms

\[
\text{person} \{\text{is-} \text{aged}, \text{diplomolevel}, \ldots\} \\
\text{student} \{\text{topic} \ldots\} \\
\text{staff} \{\text{beginner}, \text{beginnerfrom} \ldots\} \\
\text{researcher} \{\text{is-} \text{aged}, \text{diplomolevel}, \text{topic}\} \\
\text{student} \_ \text{researcher} \\
\text{researcher} \_ \text{isa} \text{staff} \\
\text{staff} \_ \text{isa} \text{person}
\]

Figure 1 describes a hierarchy of classes, with method-names. Corresponding to it we would have the following:

\[
\ldots \text{researcher} \_ \text{isa} \text{researcher} \text{isa} \text{student} \text{researcher} \_ \text{isa} \text{student} \\
(43)
\]

\[
\ldots \text{pat instance researcher} \\
\text{ida instance researcher} \\
\text{franz instance student} \_ \text{researcher} \\
\text{age(ida,20)} \\
\text{age(pat,35)} \\
\text{age(franz, no_value)} \\
(47)
\]

\[
\ldots \text{researcher with X\_in\_aged(Y): age(X,Y), Y/= no_value} \\
\text{researcher with X\_diplomolevel(Y): dip(X,Y), Y/= no_value} \\
\text{staff with X\_beginner: X\_diplomolevel(Y), Y<=4} \\
\text{X\_experiencelevel(Z), Z<=3} \\
\ldots \\
\text{person with X\_in\_aged(Y): askuser(age,X,Y)} \\
\text{person with X\_diplomolevel(Y): askuser(diplomolevel,X,Y)} \\
\text{researcher with X\_topic(Y): X\_in\_team(Z), topics(Z,Y)} \\
\text{student with X\_topic(Y): record(X,Y)} \\
(51)
\]

Examples in this paragraph will be taken from Figure 1 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 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

\[
?\_\text{franz\_in\_aged(Y)} \\
\text{Such a query in an object programming system with call/return features, would yield as answer a free variable as it would be deemed to have had an answer given at class researcher where the procedure is inaged is defined (but fails in the Logic Programming sense), hence yielding the free variable answer or perhaps an error. To palliate this a send super call, classical in object systems, could be used in the body of is_aged, but this is highly procedural in nature and thus should probably not be used in this context. It would present other problems as well (see next). Another more problematic example would indeed be}
\]

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

4.3. Default Methods

Let us now assume the query

\[
?\_\text{pat\_in\_aged(Y)} \\
\text{with S/F (success/failure) as the call mechanism semantics. This would yield a first answer "35" using data of Figure 1 and when backtracked over, would call for 'askuser', an obviously non-desirable feature. Prolog programmers would thus rewrite [r1] as}
\]

\[
?\_\text{researcher with X\_in\_aged(Y): age(X,Y), Y/= no_value} \\
(58)
\]

756 / ENGINEERING
which solves this problem, by stopping propagation. But then, what about the query 

'?:Whoisanaged(Y). X 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):- body2 (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],[2] - 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. I'hus 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 answer (i.e succeeds) a deterministic method stops all calls for the same object when backtracking 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, ica is a researcher and the path to (r9a) is cut in (r7a); as ica 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) fail for franz; even though franz is also a student, (r8) will not be tried as (r9a) is deterministic.

?-joe:topic(Y) Y=knowledge-bases and Y=logic, could be given by (r7a) and (r9a), assuming joe 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 = joe, Y = knowledge-bases
X = john, Y = logic
X = franz, Y = decision-making
X = ida, Y = knowledge-bases

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 lists couples (an instance of researcher, a child) lists instances of researcher for whom X:has_children(Y) succeeds: Y is hidden

laocwcs (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)), 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:aged(Y), tally(Y,Z), aveg(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 performance.

Finally, there is another available set of evaluable operators which allows the schema to be queried, i.e. the set of declarations itself on as to give meta-level 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 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 language 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 benefited from discussions with the Logic Programming group of ECRC, especially David Chan and Reinhard Enders.

REFERENCES