Mixing Rules and Objects : An Experiment in the World of Euclidean Geometry

François Pachet

RAPPORT LAFORIA N° 19/90, Juillet 90

This paper was submitted to the ISCIS V conference, Turkey 90.

Mixing Rules and Objects : An Experiment in the World of Euclidean Geometry

François Pachet

Laforia, Université Paris VI, Tour 46-00, 4 Place Jussieu
75252 Paris Cedex 05
fdp@litp.ibp.fr

Abstract
Objects are often used in Knowledge Representation systems because they provide powerful means for representing hierarchically both *structures*, and modular *behaviors*. On the other hand, production systems provide essential *deductive* mechanisms which are useful for representing certain kinds of reasoning.
Combining rules and objects create rich and complex environments in which the capacities of both representation schemes interact with each other in many ways.
We describe here an experiment made in order to investigate those interactions. We experimented with the system Opus [7], which adds OPS5-like first order rules to the standard Smalltalk environment.
The testbed for experimenting with these techniques is a subset of Euclidean Geometry : the system finds characteristics and properties about geometrical objects defined by the user through a graphical interface, allowing interactive modifications and instant reaction of the system.
The expertise of the system consists in a set of rules which infer properties about objects in a "Euclidean" way, i.e. without performing real computations and by building intermediary objects along the reasoning .
We will first describe the system and its environment, and then describe several extensions : to objects, by introducing viewpoints; to rules by adding recursive call in premise position; and to control by allowing meta-rules.
Finally, we give some examples of things the system cannot represent and handle, and suggest future extensions.

1 Introduction

Mathematical reasoning has long been a favorite field for experimenting with reasoning techniques and representation systems. We can split those representations systems roughly into two different approaches :

- Formal ones, based on predicate calculus or first order logic (and their extensions) which provide formal systems in which it is possible to infer general theorems from axioms.
- Informal ones, which provide deductive mechanisms not related to any formal system or logic, but which give more natural representation capabilities.

We followed the second approach, and moreover focussed on representing *instantiated* reasonings (i.e. handling "real" or existing objects). The aim of the system is to experiment with an environment in which it is possible to teach the system, in a way or another, our own clues and hints about typical reasoning situations.
Following the same approach, [10] has written a system called Muscadet to represent mathematical reasoning, which is able to prove and find theorems from an initial set of forward chaining rules; [1] describes a system for representing geometric constructions with an object-oriented environment to which he added a semantic network able to classify automatically objects, according to constraints specified at the class level.

2 Geometrical objects

2.1 Limitations of class inheritance taxonomies

The declarative aspect of Object Oriented Languages like Smalltalk is a highly controversial topic. Having the possibility of defining a class taxonomy by inheritance of

properties and structures is as powerful as it is frustrating : the fact, for instance that class *Square* cannot be described as a subclass of Rectangle, for which *two adjacent sides are of equal length*, is a starting point of our investigations.
The intrinsic limitations of class inheritance, (mainly subclassing by adjonction of attributes) make whole parts of human cognition difficult to represent. Geometric classification is one example of structures that standard class-based languages cannot represent.
Moreover, although inheritance may be considered a kind of deductive mechanism, the procedural description attached to classes by methods does not allow for representing reasonings such as the ones described here. The Object-Oriented paradigm lacks a real deductive facility.

2.2 Opus

Mixing rules and objects is not new and has already been experimented in a number of systems including commercially available ones (Art, Kee, Knowledge Craft, Humble, Oks [12]). Opus [7] consists in the standard Smalltalk environment to which are added forward chaining production rules, and is the only one to combine a fully-fledged Object-Oriented environment with a first order inference engine (Humble provides only 0+ order rules; Kee, Art, Knowledge Craft offer reduced "frame-oriented" environments), thus offering maximum flexibility and power in the representation.

Opus rules are OPS5 rule transposed in the Smalltalk environment. The rules are organized into various protocols (like Smalltalk methods), such as parallelograms, triangles, squares; each of which consisting in a set of rules. A rule consist of a name, a declaration part, any number of premises and a conclusion. A premise is any Smalltalk expression, supposedly giving a boolean result, and in which variables can be used. The variables used in the rule are declared in the declaration part by specifying for each of them the class of the objects it can match.
a typical and simple example taken from the geometric ruleBase is :

```
triangle

        | Polygon p |

        p listOfPoints size = 3.
        p isNotA: Triangle.

        actions
        p addType: Triangle.
        p modified.
```

When triggered, this rule will bind variable p to any object of class Polygon, having 3 points, for which the type Triangle has not yet been found, and assign it the "Triangle" type. The object denoted by the variable p is then reported *modified* to the system so that subsequent retriggering of rules may be done.
The rules are compiled using an extension of the Rete algorithm [3] to handle multi-input nodes.
Our system, called NeOpus consists in several extensions added to Opus, and described in the next sections.

2.3 Domain

The domain chosen is a subset of Euclidean Geometry, and was chosen for its easy access, acceptable level of complexity, and widely disseminated expertise. Adding new rules may be done by any one interested in basic Euclidean Geometry, making the system open to a large audience.
More over, it was possible, thanks to the versatility of Smalltalk's interface to implement easily a graphical interface, which an instant reaction of the system.

3 First Step : Writing rules in Opus

3.1 Types

The basic objects of the system, on which reasonings are made, are defined by a Smalltalk class : *GeometricalObject,* having no attribute (or instance variable (i.v)). Instances of this class are created graphically by the user, and submitted to the inference engine as material for thought.
The main task of the rule base will be to find out the most specific types (i.e. Rectangle, Square, IsoscelesTriangle ...) of a given geometricalObject. Other tasks are considered, such as finding properties between several objects, but are not described here.

To encode the natural classification of those types, we represented them by classes of a special kind (subclasses of class *Type*), in order to represent the multiple hierarchy links between them.
We built our "hand-made" classification tree without using the multiple inheritance mechanism of Smalltalk because we did not want the i.v. to be systematically inherited by all subclasses.
Following the "programming by metaclasses" technique advocated by [2, 9], we implemented types using a description at the metaclass level : Metaclass *Type class* has thus one i.v. *superTypes,* which contains the list of its direct super types (in the geometric sense). Every subclass of Type has thus a particular value for its i.v. superTypes, thanks to the parallel inheritance of metaclasses [4, 2] . Method *allSuperTypes* computes the list of all superTypes (up to root type *Type*).

Subclasses of *Type* include class *Polygon*, having a single i.v. *listOfPoints*, containing the list of its points, which are themselves instances of the standard Smalltalk class *Point*. Class *Polygon* defines methods for building associated *segments* linking two of its points (diagonals, medians, mediatrices), displaying and interfacing with the system.
Class *Square* for example is a subclass of *Type*, whose *superType* list is : (*Rectangle*, *Lozenge*).

```
Type define: #Square
        superTypes: #(Rectangle Lozenge)

Square superclass-> Type
Square superTypes -> (Rectangle Lozenge)
Square allSuperTypes -> (Rectangle Trapezoid IsoscelesTrapezoid
        RightAngledTrapezoid Lozenge Polygon)
```

In this model, segments are represented by instances of class *Segment* (subclass of *Type*, without superTypes), having two i.v. (*pointA* and *pointB*) containing its 2 end-points.
Though segments have indirect access to the cartesian coordinates of their two points, there is no method to compute their length (it generally involves a square root computation, which is inherently approximate) except in those cases when the segment is horizontal or vertical (the length is then simply the subtraction of its two points).

```
Type define: #Segment
        superTypes: #(Type)
        ivn: 'pointA pointB'

length
self isHorizontal ifTrue: [†(pointA x - pointB x) abs].
self isVertical ifTrue: [†(pointA y - pointB y) abs].
†self error: 'I cannot compute my length'

isHorizontal
†pointA y = pointB y

isVertical
†pointA x = pointB x
```

3.2 Viewpoints

Linking objects to their types is made by using a *viewpoint* mechanism. In this scheme, an object can be seen under different perspectives, depending on the context of the message call. In our system, for instance, a geometrical object can be seen as a simple triangle (for which we have access to its three points), or a RectangleTriangle, for which we have access to a *hypothenuse* or a corner. Finding properties for an object will consist in creating new viewpoints for it.
Other object-oriented systems extend objects to viewpoints, such as [5] with perspectives, and [13] with activities, but do not include them in any deductive mechanism. On the other hand, this approach extended to coreference is used as a central deductive mechanism in the system Madeleine [11], but in a backward-chaining environment.

In our implementation, each geometricalObject has access to its viewpoints through a Dictionary (i.e. a list of pairs key/value).
Method *as*: allow delegation [8] to a particular viewpoint of the object, and *isA:* and *addType:* give access to the viewpoints.
Class GeometricalObject is thus defined as :

```
Object subclass: #GeometricalObject
        instanceVariableNames: 'viewPoints'
        classVariableNames: ''
        pooldictionaries: ''

!Segment methodsFor: 'viewpoints accessing!

as: aClass
†viewPoints at: aClass ifAbsent:
[†self error: 'I have no viewPoint of this type now'].

isA: aClass
†viewPoints keys includes: aClass

addType: aClass
viewPoints at: aClass put: (aClass new)

addType: anObject asA: aClass
viewPoints at: aClass put: anObject
```

This implementation of viewpoints allows geometrical objects to be seen under different perspectives, but only allows one viewpoint of a given class to exist. This is due to the access mode to viewpoints : explicit delegation (method *as:*) makes it impossible to have multiple viewpoints for a given object. For instance, an EquilateralTriangle could be defined as a Triangle which is Isosceles for each of its points : our representation can not handle this since only one viewpoint of class IsoscelesTriangle would exist for a given Triangle.
With this basis, a number of rules have been written, allowing for instance to decide that a polygon is a Rectangle, if it has two horizontal sides, and two vertical ones; or concluding that an object which is both a Lozenge and a Rectangle is a Square.

4 Second Step : recursive calls

In a number of cases, geometric reasoning leads to building intermediary objects, which may be useful for the current proof. For example, one may prove that a triangle ABC is isosceles by building the projection of a vertex (A) onto the base (BC), and then check that it is equal to its middle. Middle of Segment BC is in our case directly computable if BC is horizontal or vertical, and its length is odd (cf Figure 1).
As a first extension to Opus, we introduced special variables in rules : intermediary objects are represented also by variables, but with a class declaration of Dummy, which means that these variables will not be matched against any existing object. They behave just like local

variables, and the corresponding assignation premise will not be considered as returning a boolean value, but rather as returning a nil/notNil expression.

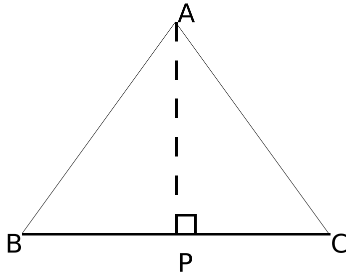

Figure 1.

```
IsoscelesTriangle

        | Triangle t. Dummy BC A P|

        (t isA: IsoscelesTriangle) not.
        t hasAnHorizontalSide.
        BC <- t horizontalSide.
        A <- t pointOpposedTo: BC.
        P <- A projectOnto: BC.
        P = BC middle.
        actions
        t addType: IsoscelesTriangle.
        t modified.
```

These intermediary objects may not only be local variables used for syntactic convenience, but also active elements for which sub goals can be created. For example, one may want to prove that a parallelogram ABCD is a lozenge by proving that a particular triangle built from it BDP (such that BP = 2 BC) is right-angled (cf Figure 2).

The actual coding for the rule is thus straightforward :

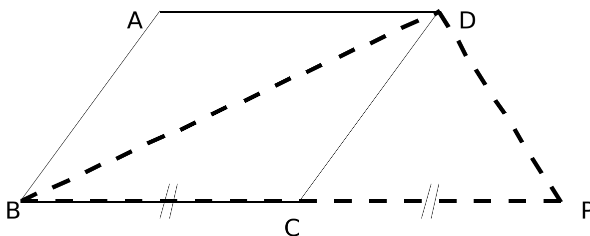

Figure 2.

```
lozengeByRightAngledTriangle
| Parallelogram t.
Dummy twoSides A B C P newObject|

(t isA: Lozenge) not.
t hasTwoHorizontalSides.
twoSides <- t twoHorizontalSides.
A <- twoSides lowest leftMost.
B <- twoSides highest rightMost.
C <- twoSides lowest rightMost.
P <- C + C - A.
newObject <- Triangle p1: A p2: B p3: P.
NeOpus
        throw: (GeometricalObject new
        addType: newObject asA: Triangle)
until:
[newObject isA: RightAngledTriangle
                and:
[(newObject as: RightAngledTriangle) vertex = B]].
actions
        t addType: Lozenge.
        t modified
```

Under certain conditions (*t hasTwoHorizontalSides*), a new geometricalObject (called *newObject*, and declared as *Dummy*), would thus be constructed out of points A, B and P (themselves declared Dummy, since depending on t, the actual object beeing matched).
This newObject is then associated to a new geometricalObject as one of its viewpoints (*GeometricalObject new addType: newObject asA: Triangle*), and then submitted to NeOpus, in order to prove that it is a right-angled triangle, of vertex B (*NeOpus throw:until:*[1]).

[1]Method name *throw*: reminds us of the etymological aspect of *objects* (ob-ject : latin "thrown before us")

This method *throw:until,* was first implemented as a recursive call to the inference engine, with a standard stacking mechanism such as :

1) save the current state (mainly the state of the conflict set),
2) perform the inference cycle until all rules exhausted (and return false) or getting a truth value for the block passed as a parameter (*until:*),
3) restore the state.

This mechanism gives the ability to make forward-chaining inferences from a dynamically created state, and to get a result back (the result of the block passed as a parameter of *throw:until*) as a premise evaluation. However this is not a procedural call nor a backward chaining call because the engine has no particular notion of a *goal* to acheive : it may endlessly wander and make inferences that are not relevant to the context in which the object was created.
For instance, let us consider the following rule :

"*A triangle ABC is proven isosceles, with base BC, if the constructed parallelogram ABCP, (P being A translated by BC) is proven to be a lozenge*".
In our example, when the preceding intermediary object is created and launched by the engine, this rule becomes candidate. However, it would not be relevant, since the current goal is to prove that this triangle is right-angled (cf Figure 3):

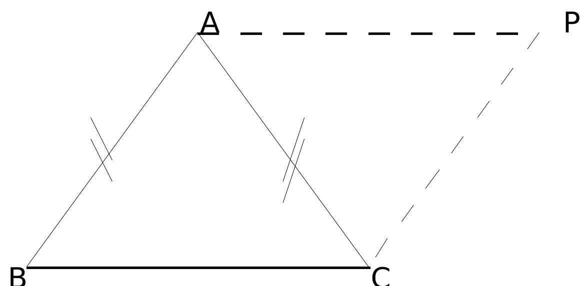

Figure 3.

```
IsoscelesTriangleByLozenge
        |Triangle t.
        Dummy BC A B C P newObject|

        (t isA: IsoscelesTriangle) not.
        t hasOneHorizontalSide.
        BC <- t HorizontalSide.
        A <- t pointOpposedTo: BC.
        B <- BC leftMost.
        C <- BC rightMost.
        P <- A + C - B.
        newObject <- Parallelogram new with: A
with: B with: C with: P.
        NeOpus throw: (GeometricalObject new
addType: newObject as: Parallelogram)
        until: [newObject isA: Lozenge].
actions
        t addType: IsoscelesTriangle.
        t modified.
```

The engine would then build new objects ad libitum, and never stop (cf Figure 4) :

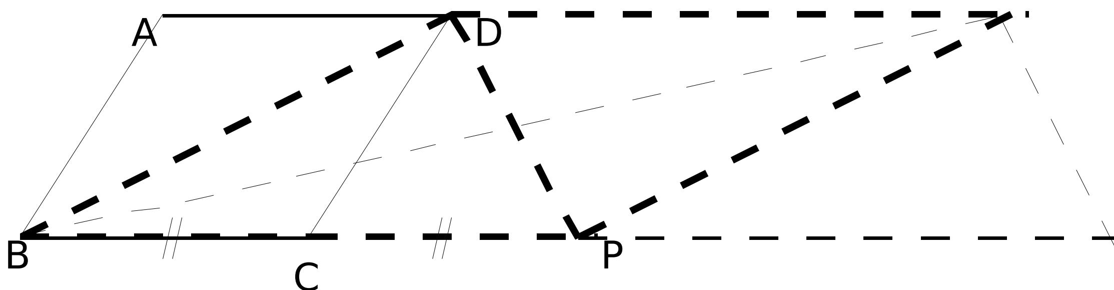

Figure 4.

This leads to the notion of *goal* described in the next section.

5 Third Step : goals and metarules

Allowing the notion of goals and sub goals in a forward chaining mechanism is not easy. Having goal objects is dangerous because it perverts the rules which must be changed to include extra premises testing for the presence or absence of those goals.
A better solution consists in handling these goals at a higher (or meta) level. The goals are objects that will, among other things, influence the choice of the rule to be fired rather than modifying the premise parts of the rules.
Opus was extended so that the control of a ruleBase could be written by rules, in a associated Rule Base (a MetaBase).

More precisely, the cycle of the engine is[2] :

1 - Send all concerned instances (in the Rete network in our case, or any other associative mechanism) to be matched against the relevant rules,
2 - Loop until all instanciations of the rules are exhausted,
3 - Give a result.

```
start
        self sendInstances; controlLoop;return
```

In order to have control over each step of the sequence, in a declarative manner, we replaced this method by a call to an associated (Meta) Rule Base. Thus each step can be correctly controlled :

step 1, allowing only concerned objects to be sent to relevant rule patterns,
step2 allowing rules to be fired preemptively according to the current context,
step3 allowing a result to be returned to the calling rule base.

A Meta Base (Rule Base2) controlling a Rule Base (Rule Base1) is itself a Rule Base. Meta rules (rules of Rule Base2) could themselves be conflicting, as soon as they are mixed with the preceding ones for instances, which could be used as default behavior for Rule Base1.
Meta Rule Bases can thus be themselves controlled by other (meta-meta) Rule Bases, and so on. The infinite regression stops when a rule base has no associated Meta Base, in which case a default procedural behavior is used.[3]
Recursive calls are now expressed as particular goals, which are handled at the meta level, and which, if successful, will retrigger corresponding premises.
There is no more stacking of inference cycles as in the first version, but rather a creation of multiple and concurrent goals which are all handled at the same meta level, allowing for a declarative control, and avoiding the consequence of stacking, mainly the impossibility of breaking the stack without breaking the calling procedure.
This approach leads to a "multilevel" programming in which not only expertise is written, but also the way to use it. It follows the *declarative* approach to Knowledge Representation since it provides an explicit representation of the interpretation of the expertise. This approach is to be compared with the system MPVC [6], which implements a general reflexive multiple viewpoint mechanism, itself described using its own representation paradigm.
Once again, all this is made possible thanks to the Smalltalk environment which allows multiple editing (of classes, rules), inspecting (of objects, conflict sets, Rete networks), and debugging.

---

[2]This sequencing is to be compared with the general sequencing mechanism of Smalltalk MVC controllers defined as :

```
controlActivity
        self controlInitialize,controlLoop,controlTerminate
```

[3]More generally, Rule Base inheritance can be described entirely at the meta level : in conflicting cases, a rule written in Rule Base B, sub-Rule Base of A, will be triggered before a rule of A

6 - Things the system cannot represent

The absence of any fact base makes it uneasy to represent assertions, predicates, or relations between objects. The representation is based on side-effects on the environment : mainly adding types to objects, and modifying their attributes. Hence, it is difficult to represent statements involving several objects, such as "*Segment AB and Segment AC have the same length*", or "*P belongs to the circle drawn round to ABC*".
As said earlier, the system only makes instanciated reasonings : it cannot infer general properties about geometrical objects, if it does not "hold" them physically.

These problems come from the fact that there is no explicit representation of Smalltalk statements : premises of rules are compiled into a Rete network, and do not exist as first class objects, just like standard Smalltalk methods (unlike in Lisp for instance, in which lambda-expressions are first class objects). However, instead of adding a "reified" representation of these statements, we tried to explore as much as possible the side-effect style of programming, avoiding higher levels of description.

This exploration is not exhausted : it is possible, in some cases, to avoid using predicates. The equality of two segments (Equals (AB, AC)) may be represented by the existence of a pair of segments : Pair (AB, AC), having the *property* of equality. These pairs of objects extend the representation vocabulary but do not add any higher level of description, and thus fit in our scheme. However, new problems arise, mainly due to the fact that objects appear several times (as themselves, and as a component of a pair), and that rules have to take them into account.

7 - Conclusion

Our system makes inferences on geometrical objects using forward chaining rules which manipulate Smalltalk objects. In order to reach a acceptable level of expression and integration, we had to make three major extensions : from objects to viewpoints, from standard premises to recursive forward chaining, from procedural control to meta rules.
The system is now able to make reasonings involving intermediary objects constructions in order to find out characteristics of geometrical objects.
However it lacks some representational power and is not able to manipulate general assertions about objects if it cannot explicitely build objects associated to them.
Future extensions will include more elaborate metabases that will have a more precise representation of goals, and that will allow, among other things, inheritance of Rule Bases.

References

1. Braun G.
Sur la programmation de constructions géométriques.
Thèse d' université. Strasbourg 88.

2. Cointe P., Briot J.P.
Programming with ObjVlisp metaclasses in Smalltalk 80.
Proceedings of OOPSLA '89

3. Forgy C. L.
Rete : A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem.
Artificial Intelligence Vol. 19 (1982) pp. 17-37.

4. Goldberg A., Robson D.
Smaltalk-80 : The Language and its Implementation.
Addison-Wesley, 1983.

5. Goldstein I., Bobrow G.
Extended Object-Oriented Programing in Smalltalk.
Lisp conference, Stanford pp. 75-81, 1980

6. Krief P.
MPVC : Un système interactif de construction d'environnements de prototypage de multiples outils d'interprétation de modèles de représentation.
Thèse d'Université, Paris VIII, June 90.

7. Laursen J. , Atkinson R.
Opus : A Smalltalk Production System.
Proceedings of OOPSLA '87 pp. 377-387.

8. Lieberman H.
Delegation and Inheritance : two mechanisms for sharing knowledge in Object-Oriented Systems.
3ième journée d'études sur les langages orientés objet, AFCET, Paris, 1986.

9. Pachet F.
A Practical Use of Metaclasses.
Proceedings of TOOLS '89, pp. 233-242, Paris.

10. Pastre D.
Muscadet : an automatic Theorem Proving System using Knowledge and Metaknowledge in mathematics.
Artificial Intelligence, vol. 38, n° 3, 1889, pp. 257-318.

11. Volle P.
Coréférence et mécanismes déductifs dans un langage de représentation par objets.
PRC-IA, journée du pôle I. Caen 15/6/89.

12. Voyer. R.
Reflex Compilation : a New Efficient Implementation Model for Systems that Use Variables and Work in a Forward Chaining Basis.
ISCIS IV, Izmir, Turkey, pp. 1135-1145, 1989.

13. Wolinski F.
Représentation de systèmes robotiques en Smalltalk , Convention IA pp. 685-699, 1990.