A Simple Technique for Handling Multiple Polymorphism

Daniel H. H. Ingalls
Mail Stop 22-Y
Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, CA 95014

Abstract

Certain situations arise in programming that lead to multiply polymorphic expressions, that is, expressions in which several terms may each be of variable type. In such situations, conventional object-oriented programming practice breaks down, leading to code which is not properly modular. This paper describes a simple approach to such problems which preserves all the benefits of good object-oriented programming style in the face of any degree of polymorphism. An example is given in Smalltalk-80 syntax, but the technique is relevant to all object-oriented languages.

Polymorphism and Messages

The object-oriented style of programming was introduced to overcome the complexity barrier of polymorphism in extensible languages. Previous attempts at extensible languages were tempting in their power to describe new fields of information, but they failed to deliver the same economy of description as system size increased. Procedures in an extensible language had to be polymorphic - in other words, they had to deal with arguments of many different types. The conventional solution to such polymorphism was to test for each type and then execute code appropriate to that case. This approach, although adequate for certain simple applications, violated basic principles of modularity, and led to a combinatoric explosion of complexity for large programs.

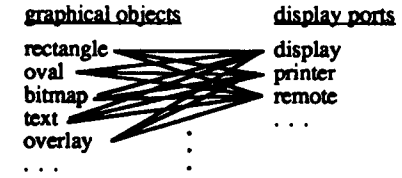
The introduction of the message paradigm for computation finally overcame this barrier, and allowed the promise of extensible languages to be realized in full. The message-sending process itself absorbs the need for type testing, and the procedures (methods), being local to their particular type (class), are not polymorphic, and do not depend on other types in the system.

All current object-oriented languages thus support simple polymorphism. That is to say, a variable or expression representing the receiver of a message may, dynamically, vary in type. Different but appropriate results will be produced, depending on the type of each receiver. This capability leads to a great simplification in the description of behavior of different but similar objects. Moreover, most object-oriented implementations provide an efficient message construct, so that this support for polymorphic receivers costs little more than a conventional procedure call.

The Problem

Certain situations arise, however, where more than one variable in an expression is independently polymorphic. Such cases usually lead to a style of coding which reverts to explicit type testing and thus brings back all the old modularity problems of procedural coding.

Let us take as an example the case of graphical objects and display ports on which such objects may be displayed. Clearly a variable holding a graphical object will frequently be polymorphic, taking on such values as rectangles, ovals, lines, text, bitmap images, or other more complex graphical objects. At the same time, however, a variable holding a display port may well also take on values of different concrete type, such as a normal display port, a printing port, a port for remote display over a communication line, and so on. Thus we have the following doubly polymorphic interaction:



In this situation, programmers will frequently write a family of methods for each graphical object of the form:

```
<Rectangle> displayOn: aPort
    aPort isMemberOf: DisplayPort
        ifTrue: ["code for displaying on DisplayPort"].
    aPort isMemberOf: PrinterPort
        ifTrue: ["code for displaying on PrinterPort"].
    aPort isMemberOf: RemotePort
        ifTrue: ["code for displaying on RemotePort"].

... and similarly for the other graphical objects.
```

At least the code is now properly distributed so that it is local to each specific graphical object, and it would be easy to add a new kind of graphical object, or edit an existing one. However, with regard to different kinds of display ports, this code will be difficult to extend or even to maintain. Of course, the methods could have been distributed through the display port classes, but then it would be complicated to extend to new graphical objects.

Thus the programmer has been let down by conventional message dispatch, which only supports polymorphism of message receivers, not of arguments as well. The code above will be seen to grow in complexity with the degree of polymorphism, and in so doing it presents a barrier to any naive programmer wishing to add a new kind of displayable object or display port. Any error in augmenting the above code fragments to deal with a new kind of object will result in failure of existing code, possibly leading to complete loss of environmental support. These are all the problems which object-oriented programming was supposed to cure.

[Some recent object-oriented systems, such as CommonLoops[1], provide for methods that are polymorphic in more than one parameter. This relieves the programmer from having to implement the solution below, but the solution proposed is an effective one for implementing such a facility.]

The Solution

Fortunately the solution to dealing with multiple polymorphism is available in all existing object-oriented languages - it is only necessary to understand the connection between polymorphism and message sending to recognize the appropriate approach. In essence, each message transmission reduces a polymorphic variable to a monomorphic one by the type dispatch inherent in message lookup. Usually (by design), only the receiver is polymorphic, and the situation is simple. However, in the doubly polymorphic example above, the first message dispatch only does half the job - the argument to the target method is still polymorphic. This suggests that another message must be sent to reduce the remaining polymorphism.

To return to our display object example, one would define a relay method in each graphical object to effect a further dispatch on the port type as follows:

```
<Rectangle> displayOn: aPort
    aPort displayRectangle: self
<Oval> displayOn: aPort
    aPort displayOval: self
<Bitmap> displayOn: aPort
    aPort displayBitmap: self
... and similarly for the other graphical objects.
```

The information gained in the first dispatch must be preserved by introducing a new family of messages specific to the graphical object types. Now one needs only to define methods for this family of messages in each of the display port classes as follows:

```
<DisplayPort> displayRectangle: aRect
    " code to display a rectangle on a displayPort "
<DisplayPort> displayOval: aRect
    " code to display an oval on a displayPort "
<DisplayPort> displayBitmap: aRect
    " code to display a bitmap on a displayPort "
    ... and similarly for the other graphical objects,

<PrinterPort> displayRectangle: aRect
    " code to display a rectangle on a printerPort "
<PrinterPort> displayOval: aRect
    " code to display an oval on a printerPort "
< PrinterPort> displayBitmap: aRect
    " code to display a bitmap on a printerPort "
    ... and similarly for the other graphical objects,

... and similarly for the other display ports.
```

This solution preserves the modularity of object-oriented programming style. If one wishes to add a new kind of graphical object, one needs never to tamper with existing code, but only to define the relay message in the new class, and the corresponding implementation methods in each of the actual

port classes. Adding a new port class is even simpler, as it amounts only to implementing the full family of **displayX:** messages.

Of course the reverse solution in which ports relayed to graphical objects would have equally good modularity properties. The choice in this case depends on a design decision as to whether the final methods belong more in the graphical object classes or in the display port classes.

The technique described above can be used to reduce higher degrees of polymorphism as well. Each subsequent message dispatch reduces a further degree of polymorphism. Fortunately, just as double polymorphism is much less common than simple polymorphism, so are higher degrees much rarer still.

Experience

The approach outlined above has proven effective in several situations beside the display example cited. One is the interaction between different event types and event handlers. Another arises in connection with logic programming where both receiver and argument of the message **unifyWith:** are polymorphic across constants, variables, terms, and other forms. A third is an experimental rewrite of the arithmetic coercion logic in the Smalltalk-80 system.

References

[1] CommonLoops: Merging Lisp and Object-Oriented Programming, by Daniel Bobrow *et al.*, Proceedings of OOPSLA '86, September 1986, Portland Oregon