Abstract

It is impossible to talk about programming “paradigms” without bringing Object-Oriented (OO) programming to mind. The question therefore arises, is there any relationship between Indexical Programming and OO Programming?

1 Objects as Filters

Explanations of OO often take as their starting point a model of computation in which objects with internal state send messages to each other. This sounds a lot like the “naïve” dataflow model of old (streams-only) Lucid [3], and in fact many simple object/message algorithms can be coded in Lucid. The basic idea is that the objects with internal state correspond to (instances of) nonpointwise functions; and that the messages are datons sent along the pipes from producer to consumer.

As a simple example, the good old running average program

$$\text{avg}(X) = \frac{T}{N}$$

where

$$T = 0 \text{ fby } T + X;$$
$$N = 0 \text{ fby } N + 1;$$

end

takes a stream of numbers and produces a corresponding stream of averages-to-date. We can think of each input or output daton as being a one-parameter message. $T$ and $N$ are then internal instance variables updated every time a message is handled. (Assume 0/0 yields an error object.)

Of course a more “idiomatic” OO treatment of averages would look a bit different. There would be separate messages for registering a new number, for accessing the current average, and for resetting the object. The messages might look like

ADD: 23
AVG
RESET

One way to capture this in old Lucid (in fact, in pLucid) is to write a filter whose input is a stream of messages of this type coded, say, as lists. The input stream might begin

[RESET] [ADD 23] [ADD 11] [AVG] [ADD 2] [AVG] [RESET] ...

in which case the output stream would begin

17, 12, ...
Here is a pLucid filter which corresponds to this object:

$$\text{avg}(M) = T/N \text{ whenever } \text{command} \text{ eq } "\text{AVG}"$$

where

command = head(M);
param = head(tail(M));
T = case command of
    "RESET" : 0;
    "ADD" : T + param;
    "AVG" : T;
end
N = case command of
    "RESET" : 0;
    "ADD" : N + 1;
    "AVG" : N;
end

Of course, a native Lucid programmer might express the same idea in a somewhat different way—say, by eliminating lists and making \text{avg} a filter with two arguments, \text{command} and \text{param}. But the correspondence is still pretty close, and the style is practical—most of the old pLucid screen editor was structured along these lines.

We can therefore find Lucid analogs of objects (namely, filters) and messages (namely, datons). We can even regard the definition of a filter as a class, and the particular calls (with particular actual parameters) as instances of the definition.

Do we have OO programming? In my opinion, not at all. We are missing the fundamental ingredient: inheritance.

For example, suppose we wanted to define a class of more elaborate average calculators, which have extra internal memory (say, to store a running maximum and minimum) and respond to extra messages to display these values. In an OO language we can declare the new class to be a subclass of the existing one, and inherit all the code for dealing with the messages already described. In pLucid, however, we have to clone the definition and hack it—by adding extra definitions and extra alternatives to the case statements. We have to go through the same procedure in more conventional OO languages (such as C), and therefore the real advantages of OO are still out of reach—even though we have (analogies of) objects, messages and classes.

## 2 The Software Version Problem

If we want to find a relationship between IP and OO, we should look first of all for some in connection between inheritance and IP.

There is in fact such a connection, but it was not uncovered in the obvious way, by adding inheritance to an Indexical language. Instead, it grew out of an (initially \textit{ad hoc}) attempt by the authors to manage a family of prototype implementations of various indexical systems.

Originally, Indexical programming meant programming in One Big Language (Lucid). Implementing IP meant implementing The Language. Soon after the publication of the first Lucid Book, however, we gave up the unilingual approach and began developing a whole family of languages and systems based on indexical logic. Indexical Lucid is still the most prominent member of this family, but not the only one. Contributions to this and recent ISLIPs give some idea of the diversity which was already apparent in the Indexical family—spreadsheets, logic programming, attribute grammars, higher-order languages, and so on.

One great disadvantage of having a family of systems is that you need a family of implementations. Each single paradigm has many variations—temporal logic programming with branching time, or with user-defined dimensions, multidimensional spreadsheets, infinite branching time, etc.
Yet there should be no need to produce separate implementations, because these systems really
are members of a family, and have strong family resemblance. Instead, we can (incrementally)
build a family of implementations in which related members share code which implements common
features.

In other words, we can view these different systems as versions (variants) of a single piece
of software. We therefore need an approach to version management and configuration which
allows code sharing between versions. Obviously, this is an important problem not limited to
software which implements indexical systems. In fact, almost all important commercial products
(computers, copiers, airplanes, automobiles, even books and newspapers) exist in versions. The
control and configuration of the related software and documentation is already a serious problem.

3 Versions as Possible Worlds

The indexical approach to version control is based on two simple observations.

The first is that a family of variants of a piece of software can be thought of as an intension,
one which varies over a context space whose possible worlds are the particular versions.

The second observation is that a piece of software is the result of combining its components,
and that this combination process is a pointwise operation. In other words, we configure the (say) french
version of a word processor by assembling the french versions of its basic components.

These observations are, by themselves, not much help, because they seem to rule out sharing
between related versions. Most likely (especially if the software is well designed) there will only
be a few modules for which the french version is different from the standard one. It is wasteful
and error prone to require the programmer to produce separate french copies of every module.

Instead, we can establish a default convention: if no separate french version of a module M
can be found, then the french version of M (the value of M at the french world) is the same as
its standard value (its value at the vanilla world). Conceptually, then, M is still an intension with
an extension at each world; but the programmer does not actually have to create separate labeled
copies of each extension.

The configuration process is only slightly more complicated. To configure the french version,
we assemble all the required modules, in each case taking the one labeled french if it exists,
otherwise taking the one labeled as standard.

The principle involved becomes clear once we introduce subversions; for example, with versions french\%quebec and french\%guadeloupe, as subversions of french, which (along with, say, german
and italian) is a subversion of standard. When configuring the french\%quebec version we try, for
each module, to find an explicit french\%quebec version. If none exists, we look for a separate french
version; and if that is not available either, we take the standard version.

4 Version Inheritance

The principle is that if there is a copy of a module M labeled with a version α, but none labeled
with immediate subversion β, then Mβ is taken to be equal to Mα. In other words, by default
the β version of the whole system inherits its components from those of version α. In general, to
configure version α of the system, we examine, for each module Mα, the set of all subversions γ
of α for which a labeled copy of M exists (we call these γ the relevant versions of M). If this set
has a least element (least in the subversion ordering) then we take that version of M; otherwise
there is an error condition. We can summarize the procedure by saying that we assemble a system
by choosing, for each module, the most relevant version of that module.

The author and John Plaice have developed a simple version management system (Lemur [2])
for C software based on the indexical approach. Lemur allows sharing of object code as well: it
keeps track of the versions used in compiling a .o file, and attaches the appropriate tag. The
system allows multiple inheritance: the sum α + β of versions is a subversion of both α and β.
Sums are used to recombine separate versions. For example, the french + fast version is the one
which is both *french* and *fast*. When combining the *french* and *fast* versions, the programmers have to consider only those modules which exist in both versions. In these cases, it is necessary to produce an explicit *french + fast* version, for otherwise configuration will fail (neither the *french* nor the *fast* will be the most relevant).

5 Program Intensions

The basic idea behind the Lemur approach is to use a partially ordered context space and an inheritance rule as the basis of an economical representation of an intension. The same idea can be applied in other systems in which the program text itself is not monolithic but varies over a context space.

In a spreadsheet program, for example, the definition of the sheet $S$ is given by a whole array of separate expressions, one for each point in the context space. And we have already seen examples where this is uneconomical because all the formulas (say, in a particular row) are identical. We can avoid this problem by introducing extra context points which stand for entire classes of cells, and allow the user to give expressions for these classes which will be inherited by the cells in each class.

For example, we can add class-contexts for each row and each column, and one for the entire sheet (with subset ordering). Then to calculate the value at a particular cell, we first check if there is an expression in that particular cell. If there is, we evaluate it, otherwise we look for an expression attached to the row or column in which the cell appears. If there is (one), we evaluate it, otherwise look for the generic (default) expression attached to the whole sheet. Clearly multiple inheritance arises again, when the classes (considered as sets of cells) overlap. We could forbid them; or we could require that at the point where they overlap, the expressions have the same value.

Context inheritance also works well with attribute grammars, where typically many productions (such as those involving arithmetic operations) all have identical definitions.

6 The Revised Analogy

In this revised analogy, the objects of OO correspond to the contexts or possible worlds of the intensional model.

For example, in the OO approach to spreadsheets, we might make each cell an object, which understands at least the message $S$, meaning evaluate the spreadsheet at that cell. In the intensional approach to spreadsheets, each cell is a possible world, and we can demand the value of the spreadsheet value $S$ at a given cell. In other words, sending the $S$ message to a given cell corresponds to demanding the value of $S$ at the corresponding world.

In this analogy, the methods of OO correspond to the defining expressions for the variables. In the OO model, we look first at the receiving object to see if there is code for the message in question. If there none at hand, we search up the inheritance hierarchy. Once we have found a method body, we execute it *at the original object.*

Correspondingly, when we are given a demand for the value of $S$ at a cell $c$, we first look for a defining expression in the cell itself. If there is none we look in the row, column and global pseudo cells—up the world inheritance hierarchy. Once we have found the relevant defining expression, we evaluate it *at the original cell.*

In other words, the two computational models—OO and extended eduction—are essentially the same, at least for simple programs. The main difference is in the data structures used: OO uses heap memory and pointers, eduction, an associative cache.
7 Indexical Objects

This analogy suggests one simple extension of OO: intensional objects. An intensional object is one which varies from context to context (in this case, the contexts themselves are objects).

In fact, most OO languages already have two intensional objects, namely super and self. There is no single super object—it is the object immediately above (in the inheritance hierarchy) the current object. And self is exactly the current object.

Now suppose we are implementing a spreadsheet OO-style, as a two-dimensional array of objects. We might often, in the methods, want to refer to the neighboring cells, without messy indices. We could do this quite easily with four intensional objects upper, lower, left and right, which denote the cell-objects immediately above, below, to the left, and to the right, respectively. If the definition of \( S \) at a given cell is \( \text{up} \ S + \text{right} \ S \), the corresponding method for the message \( S \) at that cell would be simply \( \text{upper} : S + \text{right} : S \).

8 Indexical OO Versioning

This indexical view of OO offers some intriguing possibilities. One big problem with OO software is versioning. The problem is that a newer version may change methods found part way up in the inheritance hierarchy. Worse yet, we may want to change the inheritance ordering in a newer version.

A possible solution lies in adapting the Lemur approach to versioning to our indexical model of OO. A message (demand) would be indexed by two coordinates: one to specify the object, the other to specify the version. Both these dimensions have an inheritance order, but it is relatively simple to combine them so that the search for methods proceeds through the new version of the hierarchy, and uses the new versions of the appropriate methods.

We can do this by making the intensional object super depend on the version coordinate as well as the object coordinate.

9 Acknowledgements

I would like to thank John Plaice and Bjorn Freeman-Benson for stimulating discussions and valuable suggestions. Weichang Du first suggested indexical inheritance, in the context of the Intensional Spreadsheet [1]. This research has been supported by a grant from the Canadian Natural Sciences and Engineering Research Council (NSERC).

References

