Object Orientation: A Mathematical Perspective

Nelson Rushton
Texas Tech University, Lubbock, Texas, USA

Abstract This paper examines language features usually associated with object orientation (OO) from the standpoint of mathematical practice and mathematical logic. Section 1 points out the occurrence of polymorphism and inheritance in mathematical theories commonly discussed as early as elementary school. Section 2 explains and illustrates the dichotomy between OO languages and common sense mathematics on one hand, and non-OO languages (such as C) and research-level mathematics on the other. This is related to the parallel dichotomy of nonmonotonic and monotonic logic used to reason about the respective kinds of systems. Section 3 discusses the claim that other aspects of OO, such as "classes", "methods", and private and public data attributes, are simply new names for old concepts, orthogonal to the core features of OO. It is pointed out that such extraneous complexity is common in the early stages of pioneering discoveries. Section 4 discusses matured, streamlined realizations of the core OO features, using the SequenceL programming language as an illustrative example. Section 5 gives a summary.

Keywords: Object Orientation, Programming Languages, Knowledge Representation

1 Core OO concepts: polymorphism and inheritance

Let us begin by considering a problem from grade school mathematics:

Problem 1
The North and South walls of a barn each consist of a 32' by 20' rectangle and an isosceles triangle with base 32' and height 12'. The east and west walls of the barn are 50' by 20' rectangles (not shown in the figure), which are perpendicular to the North and South walls. What is the volume of the interior barn?

Figure: barn as viewed from the North.

Part of the mathematical knowledge presumably used to solve this problem is given by informal propositions 1-5 below.

1. area is a function from plane figures to real numbers.
2. Triangles are plane figures.
3. If \( t \) is a triangle, then \( \text{area}(t) = 0.5 \cdot \text{base}(t) \cdot \text{height}(t) \).
4. Rectangles are plane figures.
5. If \( r \) is a rectangle, then \( \text{area}(r) = \text{width}(r) \cdot \text{height}(r) \).
6. The volume of a prism is the area of one of its faces times the distance between the two faces.

Though they might not use the terms 'function', 'real number', etc., most Americans are practically familiar with the knowledge represented in these propositions by about 5th grade. Few 5th graders have heard of "object orientation", but I claim they are practically familiar with that too -- since it is part and parcel of their understanding of Propositions 1-5. The following paragraphs explain why.

A professional mathematician might say that a plane figure is a Lebesgue measurable set of points in \( \mathbb{R}^2 \), its area is its Lebesgue measure, and the formulas for areas of simple shapes such as rectangles and triangles are theorems that follow from these definitions. Or she might say something slightly different, but nothing she might say resembles the perspective of a 5th grader. To a 5th grader, a plane figure is basically this: a triangle, a rectangle, a circle, and, potentially, other kinds that we haven't got to yet.

We should not look down our nose at the 5'th grade perspective, because it is almost everyone's perspective. A scientist or engineer's list may be longer, including, perhaps such things as an ellipse, the region between the graphs of two functions, and the region bounded by a closed curve with a given parametric formula, but it still ends with dot-dot-dot's, and in that sense it is essentially more like a school child's than...
2 Polymorphism from the standpoint of mathematical logic

For the sake of argument, imagine we wanted to express 5th grade mathematics -- and just 5th grade mathematics -- in the paradigm of contemporary research mathematics. At that level, contemporary mathematical discourse consists of theorems, proofs, and definitions, with a handful of set-theoretic axioms in the background as are part of a shared culture. In this context, having already defined triangles and rectangles, the closest we could come to the effect of Propositions 2 and 4 is as follows:

**Definition A [plane figure]:** A plane figure is a triangle or a rectangle.

Note this definition is not open to extension, as Axioms 2 and 4 would be. Similarly, in the contemporary research mathematics paradigm, Propositions 1, 3, and 5, would combined into a closed (i.e., once and for all) definition of the area function on plane figures, such as

**Definition B [area]:** If \( p \) is plane figure then \( \text{area}(p) \) is \( 0.5 \cdot \text{base}(p) \cdot \text{height}(p) \) if \( p \) is a triangle, and \( \text{width}(p) \cdot \text{height}(p) \) if \( p \) is a rectangle.

The "closedness" of the conceptions mathematicians work with has the prima facie disadvantage that it is not open to elaboration. If we wanted to add circles to the theory, for example, we would need a new term for the union of circles, triangles, and rectangles; we could not use plane figure because it is already defined by Definition A.

On the other hand, the advantage of the closedness of Definition A is that it determines the theorems that hold about plane figures. For example, from Definition A follows

**Theorem C:** Every plane figure is a polygon.

In the context of this paradigm, the theorem will remain true in any extension of the discourse, just as usual in mathematical reasoning. It would be strange indeed to see a theorem stated on, say, page 24 of a math text, only to see it overturned by a contradictory theorem on page 42. Technically, we may say that the logic of mathematical discourse is monotonic: if a theory \( T \) entails a theorem, then any extension of \( T \) will also entail the theorem.

The logic of programs in non-OO languages, for example the C language, is monotonic like the logic of research mathematics. Because C data types are defined in closed and final fashion, i.e., without the possibility of extension, generalizations about them will remain true no matter how the containing program is extended. We cannot say the same of,
say, C++. For example, the two-line below program by itself has the property that every animal has at least two legs:

```cpp
class Animal {public: virtual int
    GetNumberOfLegs() = 4; }

class Duck : public Animal { public:
    int GetNumberOfLegs() { return 2; } }
```

But the addition of the line below breaks the theorem:

```cpp
class Fish : public Animal {public:
    int GetNumberOfLegs() { return 0; } }
```

The reason for the nonmonotonicity in the logic of OO programs is straightforward: when a theory allows the introduction of new objects of an old kind, one cannot in general expect old theorems to continue to hold. This indicates that the OO features of polymorphism and inheritance are not syntactic sugar or mere conveniences (or inconveniences), but fundamental language features with a categorical impact on the logic of the designed systems.

Incidentally, fifth grade mathematics is both extensible and monotonic -- because, technically from the standpoint of mathematical logic, its axioms have no particular intended model. In layman's terms, the very meaning of plane figure in fifth grade math is open ended. A programming language, on the other hand, cannot have it both ways: it must choose between extensibility and monotonicity, because it does have a particular intended model, namely, the behavior of executed programs. In other words, school children can work with incomplete information in ways that programs typically cannot. An average tenth grader may not be able to compute the area under the graph of the parabola $10 - x^2$, but he will strongly suspect that it is a real number greater than 1. In contrast, if we call `GetNumberOfLegs()` on a member of a subclass of `Animal` that has not yet been declared, it will crash as hard if we had called it with the argument 42.

3 Peripheral OO concepts

Besides inheritance and polymorphism, jargon associated with object orientation includes terms such as `class`, `object`, `method`, and `public` and `private` attributes. Unlike inheritance and polymorphism, these seem to be nothing but jargon: `class` is another name for a type; `objects` is another name for data, or members of types; `method` is another name for a function or procedure; and `public` vs. `private` attributes are a manifestation of visibility features (i.e., `visibility` in C) that exist in most mature languages whether they are "object oriented" or not. Certainly, using an infix dot for "method" calls (which are really function calls) is nothing but a surface feature.

Ironically, it seems the early pioneers of object orientation did not allow existing terminology (type, function, visibility, et. al.) to be polymorphically inherited in their new extensions of program semantics!

Before we decry this irony and roll our eyes at the early pioneers of OO, keep in mind that it is usual for transformative ideas to be expressed clumsily in the first few decades after their discovery -- and in some cases the first few centuries. As an example, consider a passage from Newton's *Principia Mathematica*, one of the greatest treatises in the history of science:

> Quantities, and the ratios of quantities, which in any finite time converge continually to equality, and before the end of that time approach nearer the one to the other than by any given difference, become ultimately equal.

The first thing one might note here is that any graduate student who wrote such a thing today would be chastised for their lack of rigor. We can suppose that in modern terms, Newton meant something like the following:

$$\text{If } x \text{ and } y \text{ are sequences of real numbers, and the limit of } |x_n - y_n| = 0, \text{ then the limit of } x \text{ equals the limit of } y.$$  

Holding aside for the moment our due reverence for genius of Newton, there are a few issues here. First, the proposition is false, (in case, for example, $x$ and $y$ satisfy the required condition but are unbounded say, $x_n = n^2, y = n$ for all $n > 0$). Second, even sequences that converge to the same limit do not "become ultimately equal", either in common sense or in any sense Newton ever defined (the sequences 1/2, 1/3, 1/4... and 0,0,0,... converge to the same limit, for example, but their respective members are never equal). Scholars debate whether Newton had a definition of limit at all; and if he did, it was a poor one by modern standards. The definition of limit, which today appears in an early chapter of every freshman calculus book, was unknown to Newton and Leibniz, the famed inventors of calculus; it was written by Karl Weierstrass about 200 years after *Principia* was published.

4 What does refined OO look like?

If we begin the history of object oriented programming as usual with Simula, it is now around 50 years old. OO is now fairly well worn territory, and, just as with calculus and logic, work has now been done to distil its key ideas and recast them in clear, simple terminology and notation. As mentioned in Section 3, I believe these core ideas are polymorphism and inheritance. In Section 1, we saw these core ideas used, seamlessly and without the usual jargon, in the context of grade school mathematics. What does that look like in a programming language?
Concise and elegant approaches to OO are exemplified, for example, by the modern procedural language Oberon, and by the functional languages Haskell and SequenceL. By comparison with, say, Java and C++, these languages are fairly obscure as yet. That is, to put it bluntly, the conception of OO by today's programming community is based in frameworks that needlessly multiply concepts and terminology, and which gather orthogonal concerns arbitrarily into the same box, by comparison with other lesser known but well studied frameworks. One goal of this paper is to contribute to correcting that situation.

To illustrate the fruits of research in the refinement of OO, the remainder of this section will show how Propositions 1-5 of Section 1 are represented in the SequenceL functional programming language. In order to obtain a complete running program, we represent additional required content, that was presumed as background in Section 1. Proposition 1, area is a function from plane figures to real numbers, can be translated into SequenceL as follows:

```sequence
area: Figure -> float;
```

Proposition 2, triangles are plane figures, along with a concrete definition of the triangle type and the types and functions on which they depend, is represented in SequenceL as follows:

```sequence
// A Point is a structure with x and
// y coordinates
Point ::= (x:float, y:float);

// The distance from point A to point B
// is defined as usual
dist: Point*Point -> float
dist(A,B) := sqrt((A.x - B.x)^2 + (A.y - B.y)^2);

// A triangle is a structure with
// attributes A, B, and C, whose values
// are points. The points must be
// noncollinear but the compiler will
// enforce that, so user beware.
Triangle:Figure  ::= (A: Point, B:Point, C:Point);
```

Instead of using Proposition 3 directly, we will compute the area of triangles by Heron's formula, because it is simpler when a triangle is given in terms of its vertices. We could, however, have used the usual formula.

```sequence
area: Triangle -> float
area(T) :=
let
  a:= dist(T.B,T.C);
  b:= dist(T.A,T.C);
  c:= dist(T.A,T.B);
  s:= (a+b+c)/2;
  in
    s := sqrt(s*(s-a)*(s-b)*(s-c));
```

Propositions 4 and 5 are formalized as follows:

```sequence
// By a 'rectangle' we mean a level
// rectangle, which is a figure with
// attributes upLeft and lowRight,
// both
// of which are points.
Figure Rect ::= (upLeft:Point, lowRight:Point);

// The area of a rectangle is its width
// times its height.
area: Rect -> float
area(R):= let
  width := R.upLeft.y - R.lowRight.y;
  height:= R.lowRight.x - R.upLeft.x;
  in
  width*height;
```

The overall effect is that Propositions 1-5 can be essentially transliterated one by one into SequenceL in fifth grade fashion, without the jargon usually associated with OO. The same could have been done in other modern languages such as Haskell or Oberon.

5. Conclusion
We have seen that the core concepts of OO, namely polymorphism and inheritance, are much lighter than the boxes they are usually sold in. Indeed they are used as early as 5th grade by average school children. We have also seen that the realization of these ideas in a programming language can be cast in an equally clear and simple fashion. This is all par for the course. Groundbreaking ideas often begin with a murky presentation, and are clarified over subsequent decades and centuries. This clarification is important. Even if it does not introduce substantively new ideas, real progress is contributed by eliminating needless redundancies and spurious associations, clarifying basic concepts and assumptions, and giving more readable notations. Object orientation has now been studied for about five decades, and much such work has already been accomplished. It may come as a surprise, however, this has had little impact in the programming profession.
I believe the truth is we should not be surprised by the sluggish progress of programming paradigms, even in the presence of clear, substantial, and ready-to-go improvements. While a programming language may be merely a tool when first learned, with sustained use it becomes much more than that: a language of thought for its user. Unlike most tools, the constructs and operations in terms of which we solve problems often become part of our identity. When a tool becomes part of a one's identity, part of them resists the intrusion of other tools for the same job, even if those tools are improvements.

Imagine, for example, that Bob is a forklift operator. He has invested years of learning and practice in his forklift skills, and they may well be a source of pride to him. For Bob, the very rumor of something categorically better than a forklift is liable to be frightening. The rumor may or may not be true, but it is natural that at least part of Bob's heart fearfully wishes it not to be true, and that this will affect Bob's words and actions on the subject. If Bob himself, or the forklift operating profession in corporate, gets to decide the issue, then it would be optimistic to expect a purely evidence based decision -- and more so the more profoundly improved is the new tool.

Perhaps, however, Bob's fears might be assuaged if he knew this: on top of his hard earned and highly lucrative forklift skills, all Bob needs to know in order to operate the revolutionary new tool, he already learned by 5th grade.

Acknowledgement
I would like to acknowledge Steve Turner of Texas Multicore Technologies, Inc., for suggesting the draft of this paper be worked into a publication.

5 References

