

Lobjcid: Objects in Lucid

Bjorn Freeman-Benson
University of Washington, Dept. of Computer Science and Eng.
FR-35, Seattle, WA, 91895
(206) 543-4226 bnfb@cs.washington.edu

April 11, 1991

Abstract. This paper describes a design (Lobjcid) for adding object-oriented programming features to Lucid. The focus of the paper is a discussion of whether objects are complex structures or just values in a user-defined algebra. After concluding that objects must be both structures and values simultaneously, the paper presents a design for Lobjcid in which variables represent streams of objects, which contain variables, which represent streams of objects, which ... etc. However, because of a circularity in the data structures, Lobjcid objects may represent an arbitrary directed graph of streams rather than a strict nesting as in traditional Lucid. Yet Lobjcid can still be implemented using the Lucid educative techniques.

1 — Introduction

Over the past decade, “object-oriented” has grown to become a major force in programming language design. Although some of this growth has been due to the belief that if something is object-oriented then it is, by definition, “good,” most of the growth has been due to the benefits that object-oriented programming, analysis, and design offer to software engineering [Booch 91]. Unfortunately, as a result of this growth in object-oriented programming, languages and systems that are not object-oriented, but still hold great promise for software engineering, have been relegated to the back burner. Functional programming and intensional programming in general, and Lucid in particular, are victims of this bias.

However, all is not lost—the features that make a language or system object-oriented are orthogonal to the basic language paradigm, be it logic, functional, imperative, or intensional. In fact, as demonstrated by [Jacky & Kalet 87] and others, object-oriented programs can be written in any language. Thus, object-oriented languages can be seen as traditional programming languages augmented with features that make writing object-oriented programs easier. This paper explores adding object-oriented features in the design of Lobjcid (L-obj-cid: *objects* spliced into *Lucid*).

And yet, before expending the effort to add object-oriented features to Lucid, it is useful to explore the motivations for this change. One obvious motivation is to “join the crowd” and ensure that Lucid gets mentioned in lists of object-oriented languages. A better reason is that adding objects to Lucid is equivalent to adding user-defined algebras. The basic implementation of Lucid includes only a few fixed algebras, such as numbers, booleans, and strings. Additional algebras cannot be added without reimplementing the interpreter. Thus, no matter how many fixed algebras a Lucid implementation supports, there will always be at least one that it does not. An object-oriented Lucid would still have a fixed set of primitive algebras, but it would also include mechanisms for defining additional first-class algebras without reimplementing the interpreter.

2 — What are objects?

Objects are encapsulated concepts. Objects encapsulate both data and operations, i.e., both state and behavior. Objects are not merely blocks of data (such as n -tuples) because objects include functions for manipulating that data. Similarly, objects are not just sets of functions because objects can include private state variables. Objects have been referred to as Abstract Data Types with encapsulation and inheritance, and yet, as we will see in section 4, ADTs and encapsulated ADTs have very different behaviors in an intensional language.

Object-oriented programming, design, and analysis use abstraction to focus on essential attributes while hiding the unessential (usually implementation-driven) details. Typically, each abstraction defines a *class* where a class is the structural and behavioral description of a set of objects. Each object is an instance of one class, known as its concrete type, and that concrete type may conform to multiple abstract types

Networks of objects (i.e., systems) are constructed through specialization and aggregation. Specialization describes the situation where one class of objects extends or specializes the behavior of another class, typically through inheritance. For example, a `3DPoint` is a specialization of a `2DPoint` because a `3DPoint` inherits the `x` and `y` coordinates and then includes an additional `z` coordinate. The specialized class is referred to as the subclass, and the more general class is referred to as the superclass. Aggregation describes the situation where one object is part of another, larger, object. For example, a `PostalCode` object could be part of a `StreetAddress` object which in turn is part of a `RolodexEntry` object. When the `RolodexEntry` object receives a request for the postal code, it delegates that request to the `StreetAddress` object, and thus is independent of the implementation of both the `StreetAddress` or `PostalCode` objects.

Object-oriented systems also support various kinds of polymorphism through inheritance and runtime binding. General polymorphism, where one implementation supports many different classes, is implemented via inheritance. Ad hoc polymorphism, where the same function has different implementations for different classes, is implemented via late binding of function calls, i.e., binding based on concrete types rather than on abstract types. Parametric polymorphism, where type information is first-class, is supported in some languages by classes that are objects themselves.

In summary, the key concepts of object-oriented programming are (1) that all the data and functions relevant to an object are localized around or within that object, and (2) that objects are black boxes with a visible external interface and a hidden internal implementation.

3 — Lobjcid

The design of an object-oriented language is a complex task as there are many issues to consider. References [Wegner 87, Deutsch 89] list a number of different choices that can be made, and the proceedings of the Object Oriented Languages, Systems, and Applications (OOPSLA) conference and European Conference on Object Oriented Programming (ECOOP) are full of papers arguing the relative merits of one design or another. Thus, the following design features of Lobjcid will be hailed by some and deingrated by others.

Classes

Objects in Lobjcid are members of classes, where a class is definition of the structure and behavior of its instances. Classes are defined as follows:


```

class class-name sub superclass-name, superclass-name, ...
  instance-variable-name;
  instance-variable-name;
  :
  expressions;
  :
end;

```

For example, a class of cartesian Points, and a class of strings that are displayed in color could be respectively defined as:

```

class Point sub Object
  x; y;
end;

class ColoredString sub String, Color
end;

```

Classes are first-class values in Lobjcid and thus classes are objects themselves.

Object creation A new object is created by the special function `new`.[†]

```
points = new( Point ) fby points;
```

Inheritance Classes may have multiple superclasses and subclasses, and thus the inheritance relation can form a directed acyclic graph. The class `Object` is the root of the inheritance graph. A subclass inherits the complete structure and behavior of each of its superclasses. A program in which multiple inheritance causes a conflict is incorrect.

Function Call Following common object-oriented syntax, functions can be called using inline, standard, or qualified syntax:

```

inline:  x = y + z
standard: x = +( y, z )    m = foobar( n, o, p )
qualified: x = y.+( z )    m = n.foobar( o, p )

```

Function Definition Functions are defined as follows:

```

function-name ( arg1-name : arg1-class,
                arg2-name : arg2-class, ...
                ) : result-class ==
  ... expressions potentially containing the ...
  ... special variable result ...
end;

```

For example, the addition of two `Point` objects is defined as:

```

+( p : Point, q : Point ) : Point ==
  result.x = p.x + q.x;
  result.y = p.y + q.y;
end;

```

(1)

[†]The `fby`, `next`, `current`, etc. keywords are the usual Lucid operators.

Multi-methods Commonly, object-oriented programming languages have used the term *method* when referring to a procedure or function. Furthermore, many object-oriented programming languages bind code to a function call based on just the class of the first argument. This is widely recognized as a defect, and thus Lobjcid uses function binding based on the classes of all arguments, commonly known as multi-methods.

A function call is bound to the “nearest” applicable function in the inheritance graph. Nearest is defined using a linearized version of the inheritance graph and a left-to-right ordering of the concrete types of the arguments [Bobrow et al. 88].

Private state Due to the encapsulation aspect of objects, the instance variables of an object are private to that object. In standard object-oriented languages, these private instance variables can only be accessed by functions defined within the object’s class. This is usually implemented by merging the instance variable name space with the function name space. In a multi-method language, such as Lobjcid, each function is defined within multiple classes and thus merging the name spaces often results in name clashes. For example, if `Point` addition were to be defined as follows, it would unclear which `x` instance variable each `x` name referred to.

```
+( p : Point, q : Point ) : Point ==
  x = x + x;
  y = y + y;
end;
```

Thus, when a Lobjcid class is defined, a special access function is defined for each instance variable. These access functions are the equivalent of:

```
x( p : Point ) ==
  result = x;
end; (2)
```

Now the `Point` addition can only be defined as in (1) on the previous page.

4 — Object Representation: First Attempt

Once the syntactic and semantic framework is defined, the actual representation of objects must be considered. Objects are elements of a user-defined algebra, and elements of other Lucid algebras are represented by blocks of memory (an integer is 32-bits, etc.), thus, as first attempt, objects are defined to be blocks of memory similar to the finite lists of pLucid [Wadge & Ashcroft 85]. A stream of objects is a stream of blocks of memory and therefore the usual point-wise and time-shifting Lucid operators ought to work. In other words, because objects are encapsulations of previously unbundled data and functions, functions both inside and outside the context of an object ought to have identical behavior. Unfortunately, they do not.

The problem is that because streams contain streams of objects which in turn contain instance variables, the “values” in these streams can be changed in two separate ways: either (i) by a stream of different objects, or (ii) by a stream of the same object with different values in the instance variables. When objects are merely blocks of memory, then case (i) is the only plausible scheme.

To see why, consider the following program:


```

stockholm.population + oslo.population;
where
  % A City is a simple object with one instance variable:
  % the population. The population starts at zero.
  class City sub Object
    population;
    first population = 0;
  end;

  % Adding a number to a City represents immigration and thus the population grows.
  % Note, however, that this plus function returns a new City object rather
  % than destructively updating the one passed to it.
  +( l : City, n : Number ) : City ==
    result = new( City );
    result.population = l.population + n;
  end;

  % The grow function represents internal growth (babies) and again the population grows.
  % Here note that the grow function returns the same City object as was passed. The
  % difference is that the population variable now has an additional defining equation.
  grow( l : City ) : City ==
    next population = population + 1;
    result = l;
  end;

  % Stockholm is a stream of Citys, each of which has a population
  % one greater than the previous City.
  stockholm = new( City ) fby (stockholm + 1);

  % Oslo is a stream of the same City, but that City has a steady
  % internal growth which was defined by the grow function.
  oslo = (new( City )).grow fby oslo;
end;

```

The `stockholm` stream represents case (i): a stream of individual `City` objects whose population values are one more than the population value of the previous element of the stream. The `oslo` stream represents case (ii): all of the elements of the stream are the *same* `City` object but that object's population increments over time. If the "objects as blocks of memory" representation were correct, then the two stream's input/output behavior should be identical—and yet they are not:

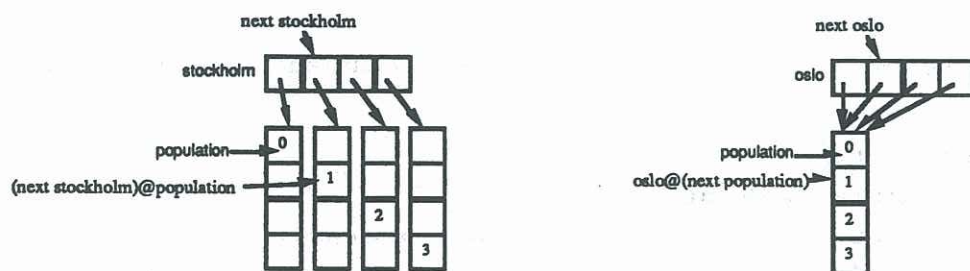


Worse, this representation does not always allow referential transparency! Therefore one must conclude that representing Lobjcid objects as blocks of memory is incorrect.

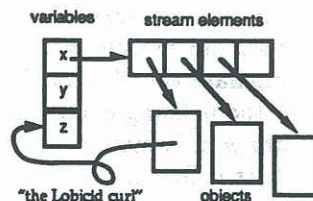
5 — Object Representation: Second Attempt

The flaw in the first attempt was that an object does double duty: it represents both a value in a user-defined algebra, and a state-holding entity. In other words, an object is different kind of entity than a pLucid list or a block of memory because a block of memory can only represent a value in an algebra[†]. Thus the instance variables of an object must themselves be full-fledged Lucid streams. This results in streams of objects containing streams of objects containing ..., a situation that Lucid users will recognize as similar to nested streams, and yet in the case of objects, the streams may not be strictly nested, i.e., the streams may form a directed graph rather than a tree.

In this second representation, the `stockholm` and `oslo` streams from the previous program have the same input/output behavior, although different internal representations:



The circularity which may be created a directed graph of streams rather than a nested tree can be illustrated by dividing the entities into three groups: variables, stream elements, and objects. In standard Lucid, variables contain stream elements, stream elements contain objects, and there it ends. In Lobjcid, objects may contain variables and thus the system is circular:



Note that this extra "Lobjcid curl" pointer can be accommodated in a standard educative Lucid implementation by adding an additional tag to each value—an object-identifier tag.

6 — Two Comments About The Semantics

6.1 — Function Semantics

In standard Lucid, a function f applied to streams is defined to mean the point-wise application of f to the elements of the stream:

[†]A large block of memory can represent a very large value indeed, but in Lucid terms it is just one element of the algebra, one value in a stream, and streams are the entities that hold state.

$$\begin{aligned}
f(< a_0, a_1, a_2, \dots >, < b_0, b_1, b_2, \dots >) &= < f(a_0, b_0), f(a_1, b_1), f(a_2, b_2), \dots > \\
&= < n_0, n_1, n_2, \dots >
\end{aligned}$$

In Lobjcid, however, functions can return streams as well as single algebra elements. Consider the x function (marked (2) on an earlier page): this function returns the stream contained in the x variable, thus point-wise application will return a stream of streams rather than a single stream:

$$\begin{aligned}
x(< C_0, C_1, C_2, \dots >) &= < x(C_0), x(C_1), x(C_2), \dots > \\
&= < < a_0, a_1, a_2, \dots >, < b_0, b_1, b_2, \dots >, \dots > \\
&\neq < n_0, n_1, n_2, \dots >
\end{aligned}$$

Thus the function call semantics must be modified to apply a **current** filter[†] to the point-wise results:

$$\begin{aligned}
x(< C_0, C_1, C_2, \dots >) &= \text{current} < x(C_0), x(C_1), x(C_2), \dots > \\
&= \text{current} < < a_0, a_1, a_2, \dots >, < b_0, b_1, b_2, \dots >, < c_0, c_1, c_2, \dots >, \dots > \\
&= < a_0, b_1, c_2, \dots >
\end{aligned}$$

6.2 — Informal Operational Model —

An informal operational model of Lobjcid is an extension of Lucid dataflow: data values flow along wires and functional components manipulate these values. However, in Lobjcid these data values are not just inanimate values—rather they are objects which may themselves contain dataflows. Lobjcid functional components can manipulate the objects either by extracting information (e.g., “+” from the example), or by modifying the defining equations (“grow” from the example). Thus one reasonable view of a Lobjcid program is as a meta-program which creates and modifies other Lobjcid programs, i.e., those programs that define the behavior of the objects. Consequently, one notes that it is possible to write a Lobjcid program containing multiple defining equations for the same data value; such programs are obviously incorrect.

7 — Conclusion —

The Lobjcid language is a paper design of how to add objects to Lucid and to intensional programming, and thus has not been implemented. However, with the exception of garbage collection, the implementation should be fairly straightforward. The important results demonstrated by this language design are (a) that objects are more than just blocks of memory, and thus more than just Abstract Data Types; and (b) that adding objects to Lucid is not a difficult task. Lobjcid updates Lucid with user-definable algebras and provides a richer programming environment.

[†]The usual Lucid current operator.

Acknowledgements

Thanks to the “anonymous” referees for their helpful comments. This research has been supported by a Chateaubriand Fellowship from the French Government, a fellowship from the National Science Foundation, the National Science Foundation under Grant No. IRI-8803294, and by the Washington Technology Center.

References

- [Bobrow et al. 88] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common Lisp Object System Specification. X3J13 Document 88-002R, June 1988.
- [Booch 91] Grady Booch. *Object-Oriented Design with Applications*. Benjamin/Cummings Publishing Company, 1991.
- [Deutsch 89] L. Peter Deutsch. The Past, Present, and Future of Smalltalk. In Stephen Cook, editor, *ECOOP'89 Proceedings*, pages 73–87, University of Nottingham, United Kingdom, July 1989. British Computer Society. Invited Paper.
- [Jacky & Kalet 87] Jonathan Jacky and Ira Kalet. An Object-Oriented Approach to a Large Scientific Application. *CACM*, 1987.
- [Wadge & Ashcroft 85] William W. Wadge and Edward A. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press, London, 1985.
- [Wegner 87] Peter Wegner. Dimension of Object-Based Language Design. In *Proceedings of the 1987 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 168–1082. ACM SIGPLAN, October 1987.