A Graph Model for Object Oriented Programming

Peter Grogono

Mark Gargul

Department of Computer Science, Concordia University
1455 de Maisonneuve Blvd. West
Montréal, Québec, Canada H3G 1M8
E-mail: {grogono,gargul}@cs.concordia.ca

Published in *SIGPLAN Notives*, **29**(7), July 1994, pages 21–28.

1 Introduction

Discussions about object oriented programming are often hindered by disagreement about basic concepts. Unlike functional programming, grounded in λ-calculus, or logic programming, grounded in logic, object oriented programming lacks a simple model that we can use as a basis for definition and discussion.

There is, of course, a large and useful body of work in which the standard techniques of semantics, based on higher order typed λ-calculus, are used to explain object oriented programming, often with emphasis on type-correctness [3, 4, 9, 10]. But there remains a lingering suspicion that these techniques, despite their power, somehow miss the point. Since object oriented programming seems to be simple and appealing to programmers, perhaps there should be a simple model that accurately describes its salient features.

But what are the salient features of object oriented progamming? They include at least object identity, local state, and dynamic binding. It is not so obvious that a model of computation should describe inheritance, because inheritance is primarily a compile-time issue. A model should be able to describe delegation, however, since delegation decisions are made at run-time.

The model should also describe the more controversial aspects of objects, such as side-effects, aliasing, the complicated forms of recursion which arise from inheritance, the use of *self* to denote the current object, and cyclic data structures. Descriptions of object oriented programming based on standard semantics need complex mechanisms to handle these phenomena, if they can handle them at all. Our model handles all of them in a natural and simple way—which is not to say that they thereby cease to be problematic!

2 Object Oriented Programming

We characterize object oriented programming in the following way. An *object* has local state and the ability to perform certain actions. The local state is defined by the values of the *instance variables* of the object. The value of each instance variable is either another object or a *primitive object* such as a boolean or an integer. Each action is called a *method*. An object may send a message to another object, requesting it to perform one of its actions: the message contains the name of a method and possibly some arguments. The binding between the name and the body of a method is created (conceptually, at least) when the object receives a message: this is called *dynamic binding*.

We are concerned primarily with systems in which each object is an instance of a particular *class*. All instances of a class respond to the same set of messages in the same way. A class defines the effect of each acceptable message either explicitly or by *inheriting* from another class.

It is natural to model an object oriented computation as a directed graph in which each vertex represents an object and each edge represents a link between objects [1, 11]. In our model, edges are labeled with the names of instance variables. Primitive values, such as integers, characters, and booleans, are represented by vertices with no out-edges.

Within this framework, we can model simple values, records, and recursive data stuctures such as lists, trees, and graphs. A record with n fields is represented by a vertex with n out-edges. We model dynamic binding by associating methods with vertices.

3 The Graph Model

The formal basis for the graph model is a collection of sets, listed below. The middle column shows the names of the sets, with a brief description on the left and some typical members on the right.

Class names	$\mathcal{C}$	$Bool, Int, Root, \alpha, \beta, \gamma$
Variable names	$\mathcal{L}$	x
Method names	$\mathcal{M}$	m, add, div
Programs	$\mathcal{P}$	P, Q, R
Vertices	$\mathcal{V}$	a, c, r, u, v, w
Integers	$\mathcal{Z}$	i, j, n

A vertex has two components: a class name and an index chosen from the integers. Thus $\mathcal{V} = \mathcal{C} \times \mathcal{Z}$ and $v = (\alpha, i)$ is a typical vertex. We define the selector “class” to extract the class from a vertex:

$$\text{class}\,((\alpha, i)) \overset{\text{def}}{=} \alpha.$$

There are two *primitive* classes, *Bool* and *Int*. It would be straightforward to add other primitive classes, such as *Float* and *String*, but they would add nothing interesting to the model. We provide abbreviations for the vertices corresponding to instances of these classes.

$$\begin{aligned} \mathbf{v}_{\text{true}} &\overset{\text{def}}{=} (Bool, 1), \\ \mathbf{v}_{\text{false}} &\overset{\text{def}}{=} (Bool, 0), \quad \text{and} \\ \mathbf{v}_{n} &\overset{\text{def}}{=} (Int, n), \quad \text{for } n \in \mathcal{Z}. \end{aligned}$$

Vertices of class *Int* are indexed by the value of the integer. For other classes, the index distinguishes instances of the class. For example, the vertices corresponding to instances of class α are $(\alpha, 0)$, $(\alpha, 1)$, $\ldots$.

A *state* is a tuple (V, E, c, a, r) in which

$V \subseteq \mathcal{V}$	is a finite set of vertices,
$E: V \times \mathcal{L} \rightharpoonup V$	is the *edge* function,
$c \in V$	is the *current* vertex,
$a \in V$	is the *argument* vertex, and
$r \in V$	is the *result* vertex.

We require the set of vertices to be finite for two reasons. First, only a finite number of vertices can be created by a finite computation. Second, no technical problems are involved in choosing an index for a new vertex.

The partial function E defines the edges of the graph. If there is an edge from u to v with label x, then $E(u, x) = v$. Otherwise, E is undefined. We use the symbol "$\dagger$" to define modified edge functions. If $E' = E \dagger (v, y, w)$ then

$$E'(u, x) = \begin{cases} w, & \text{if } u = v \text{ and } x = y, \\ E(u, x), & \text{otherwise.} \end{cases}$$

We write Σ_t to denote the set of states of the form (V, E, c, a, r). We add to this set the *non-observable* state, !. Failure of any kind, including failure to terminate, yields this state. The set of all states is $\Sigma \overset{\text{def}}{=} \Sigma_t \cup \{!\}$. We write $S.r$ to abbreviate "r, where $S = (V, E, c, a, r)$", and similarly for the other components of the state.

4 Programs and Their Meanings

A *program* $P: \Sigma \to \Sigma$ is a total function on the set of states. If $P(s) = !$, we say that P *fails* in state s. For all programs, $P(!) = !$. If P does not fail, then $P(s) = (V, E, c, a, r)$. The *result* of the program is the vertex r.

The set $\mathcal{P} \subseteq \Sigma \to \Sigma$ consists of all programs that can be generated inductively using the rules below. The first line defines *simple programs* and the second line defines *compound programs*. We use parentheses to disambiguate compound programs when necessary.

$$P \longrightarrow \mathsf{skip} \mid \mathsf{loop} \mid \mathsf{true} \mid \mathsf{false} \mid n \mid \mathsf{self} \mid \mathsf{arg} \mid \mathsf{new}\ \alpha \mid x \mid \mathsf{store}\ x$$

$$P \longrightarrow P; P \mid P\ \mathsf{else}\ P \mid \mathsf{while}\ P\ \mathsf{do}\ P \mid m(P)$$

We use metabrackets, $[\![\cdot]\!]$, to distinguish the textual representation of a program from its meaning. For example, the meaning of the program skip is $[\![\mathsf{skip}]\!]$. Table 1 uses this convention to define the meanings of simple programs.

The program skip is the identity function. The program loop always fails; it is more useful for theoretical discussions than for actual programs. The next group of programs, true, false, n, self, and arg are expressions: they yield a state S in which the result vertex $S.r$ has a particular value.

Evaluation of an instance variable, x, uses the edge function E to obtain the result $E(c, x)$. If $E(c, x)$ is undefined, the value of $[\![x]\!](V, E, c, a, r)$ is !. Conversely, the program store x yields a state in which $E(c, x) = r$.

$$
\begin{array}{lcl}
[\![\mathsf{skip}]\!](V,E,c,a,r) & = & (V,E,c,a,r) \\
[\![\mathsf{loop}]\!](V,E,c,a,r) & = & ! \\
[\![\mathsf{true}]\!](V,E,c,a,r) & = & (V \cup \{\mathbf{v}_{\mathrm{true}}\}, E, c, a, \mathbf{v}_{\mathrm{true}}) \\
[\![\mathsf{false}]\!](V,E,c,a,r) & = & (V \cup \{\mathbf{v}_{\mathrm{false}}\}, E, c, a, \mathbf{v}_{\mathrm{false}}) \\
[\![n]\!](V,E,c,a,r) & = & (V \cup \{\mathbf{v}_n\}, E, c, a, \mathbf{v}_n) \\
[\![\mathsf{self}]\!](V,E,c,a,r) & = & (V,E,c,a,c) \\
[\![\mathsf{arg}]\!](V,E,c,a,r) & = & (V,E,c,a,a) \\
[\![\mathsf{new}\ \alpha]\!](V,E,c,a,r) & = & (V \cup \{v\}, E, c, a, v) \\
 & & \quad \text{where } v = (\alpha, \mathrm{next}(\alpha, V)) \\
[\![x]\!](V,E,c,a,r) & = & (V,E,c,a,E(c,x)) \\
[\![\mathsf{store}\ x]\!](V,E,c,a,r) & = & (V, E \dagger (c,x,r), c, a, r)
\end{array}
$$

Table 1: Meanings of simple programs

The program new α introduces a new vertex into the graph. The new vertex is (α, i), in which i is a unique index for the class α. The function next, defined by

$$
\mathrm{next}(\alpha, V) \stackrel{\mathrm{def}}{=} \begin{cases} 0, & \text{if } \{\, j \mid (\alpha, j) \in V \,\} = \emptyset \\ 1 + \max\{\, j \mid (\alpha, j) \in V \,\}, & \text{otherwise.} \end{cases}
$$

chooses an index that is zero for an unpopulated class and larger than the maximum index of a populated class. It is harmless, but not useful, to use new with *Int* or *Bool* as its argument.

Table 2 defines the meanings of compound programs. Sequences are defined by functional composition. The effect of P else Q is either P, or Q, or !, depending on whether the result component of the previous state was $\mathbf{v}_{\mathrm{true}}$ or $\mathbf{v}_{\mathrm{false}}$, or some other value. Following convention, we define while P do Q as a least fixed point. The existence of the fixed point depends on an ordering of programs that we discuss elsewhere [6].

The program $m(P)$ "sends a message" to the result vertex, r_0. The model uses a partial function $\Phi\colon \mathcal{V} \times \mathcal{M} \to (\Sigma \to \Sigma)$ to obtain the program corresponding to the method name m, providing dynamic binding. We can interpret the denotation of $m(P)$ operationally as follows.

1. The initial state is $(V_0, E_0, c_0, a_0, r_0)$, in which r_0 is the "receiver" of the message.
2. The argument P is evaluated in the initial state, yielding a new state $(V_1, E_1, c_1, a_1, r_1)$, in which r_1 is the value of the argument.
3. The class of the receiver is $\alpha = \mathrm{class}\,(r_0)$.
4. The method invoked is $\Phi(\alpha, m)$. This method is evaluated in the state $(V_1, E_1, r_0, r_1, r_1)$, in which the current object, r_0, is the receiver and the argument, r_1, is the value of P. Evaluating the body yields the new state $(V_2, E_2, c_2, a_2, r_2)$.

$$
\begin{array}{lcl}
[\![P;Q]\!]s & = & [\![Q]\!]([\![P]\!]s) \\
[\![P \ \mathsf{else}\ Q]\!](V,E,c,a,r) & = & \begin{cases} [\![P]\!](V,E,c,a,r), & \text{if } r = \mathbf{v}_{\text{true}}, \\ [\![Q]\!](V,E,c,a,r), & \text{if } r = \mathbf{v}_{\text{false}}, \\ \mathbf{!}, & \text{otherwise.} \end{cases} \\
[\![\mathsf{while}\ P\ \mathsf{do}\ Q]\!]s & = & \mu X.[\![(P;\ ((Q;X)\ \mathsf{else\ skip}))]\!]s \\
[\![m(P)]\!](V_0,E_0,c_0,a_0,r_0) & = & (V_2,E_2,c_0,a_0,r_2) \\
& & \quad \text{where} \quad (V_1,E_1,c_1,a_1,r_1) = [\![P]\!](V_0,E_0,c_0,a_0,r_0) \\
& & \quad \text{and} \quad\ \ (V_2,E_2,c_2,a_2,r_2) = \Phi(\mathsf{class}\,(r_0)\,,m)(V_1,E_1,r_0,r_1,r_1)
\end{array}
$$

Table 2: Meanings of compound programs

5. The final state is $(V_2, E_2, c_0, a_0, r_2)$, in which the values of the current object and the argument have been restored from the initial state.

Message passing is not well-defined if evaluation involves infinite recursion. A fully formal treatment requires a fixed-point definition to cover this case.

In principle, we could represent the natural number n as a chain of $n+1$ vertices linked by n edges. We choose to avoid the introduction of numerous trivial methods by including integer and boolean operations in the mode. If α is a primitive class, such as *Bool* or *Int*, then $\Phi(\alpha, m)$ returns a function that implements the appropriate operation. Consider, for example, the message $(2; add(3))$. Evaluation requires $\Phi(Int, add)$, which is a primitive method of the class *Int*, defined by

$$
\Phi(Int, add)(V,E,c,a,r) \overset{\text{def}}{=} \begin{cases} (V \cup \{\mathbf{v}_{m+n}\}\,, E, c, a, \mathbf{v}_{m+n}), & \text{if } c = \mathbf{v}_m \text{ and } a = \mathbf{v}_n, \\ \mathbf{!}, & \text{otherwise.} \end{cases}
$$

Primitive classes may also contain user-defined methods. If, for instance,

$$
\Phi(Int, avg) \quad = \quad [\![\mathsf{self}; add(\mathsf{arg}); div(2)]\!]
$$

then evaluation of the message $(4; avg(6))$ would give

$$
\begin{array}{lcl}
\Phi(Int, avg)(V,E,\mathbf{v}_4,\mathbf{v}_6,r) & = & [\![\mathsf{self}; add(\mathsf{arg}); div(2)]\!](V,E,\mathbf{v}_4,\mathbf{v}_6,r) \\
& = & [\![add(\mathsf{arg}); div(2)]\!](V,E,\mathbf{v}_4,\mathbf{v}_6,\mathbf{v}_4) \\
& = & [\![div(2)]\!](V \cup \{\mathbf{v}_{10}\}\,, E,\mathbf{v}_4,\mathbf{v}_6,\mathbf{v}_{10}) \\
& = & (V \cup \{\mathbf{v}_{10},\mathbf{v}_5\}\,, E,\mathbf{v}_4,\mathbf{v}_6,\mathbf{v}_5)
\end{array}
$$

The following table gives validity conditions for both simple and compound programs. In the validity condition for $m(P)$, the subscripts correspond to the subscripts in the semantic definition.

Program	is well defined iff:
$[\![x]\!](V,E,c,a,r)$	$E(c,x)$ is defined
$[\![m(P)]\!](V_0,E_0,c_0,a_0,r_0)$	$[\![P]\!](V_0,E_0,c_0,a_0,r_0) \neq \mathbf{!}$
	and $\Phi(\mathsf{class}\,(r_0)\,,m)(V_1,E_1,r_0,r_1,r_1) \neq \mathbf{!}$

We construct complete programs according to the syntax

$$Program \longrightarrow \{ \ \mathsf{class} \ Iden \ \{ \ \mathsf{method} \ Iden \ P \} \ \}$$

in which $\{\cdots\}$ stands for "zero or more occurrences of $\cdots$". From the program and the definitions for primitive classes, we obtain a value for Φ. We assume that the program contains an entry point given by method *main* in class *Root*. The result of evaluating the program is the state

$$[\![main(0)]\!](\{v\}, \{\}, v, v, v)$$

in which $v = (Root, 0)$. Only the first occurrence of v affects the evaluation; the others are included simply to provide a well-defined initial state.

Several kinds of program depend on the result of the previous computation. These programs will normally be used as the second component of a sequence. We can improve the readability of the formal language by defining "macros" such as these:

$$\begin{aligned} x := P &\overset{\text{def}}{=} P;\ \mathsf{store}\ x, \\ \mathsf{if}\ P\ \mathsf{then}\ Q\ \mathsf{else}\ R &\overset{\text{def}}{=} P;\ (Q\ \mathsf{else}\ R), \\ P\,.\,m(Q) &\overset{\text{def}}{=} P;\ m(Q). \end{aligned}$$

5 Inheritance

In this section, we outline a simple way of introducing inheritance into the model. It is necessary to change only the binding mechanism for method names.

We define a partial order, $\preceq$, on class names. If $\alpha \preceq \beta$, we say "class α inherits from class β". This definition is reflexive: every class inherits itself. We use $\prec$ to denote "proper" inheritance, between distinct classes. We introduce a function Ψ which generalizes Φ by returning a set of methods:

$$\Psi(\alpha, m) = \{ \Phi(\gamma, m) \mid \gamma \in \Gamma \}$$

where Γ is the set of classes that satisfy these conditions:

$$\begin{aligned} \gamma \in \Gamma \text{ iff } \quad & \Phi(\gamma, m) \text{ is defined, and} \\ & \alpha \preceq \gamma \text{ and} \\ & \forall \beta \in C\,.\, \alpha \preceq \beta \prec \gamma \Rightarrow (\Phi(\beta, m) \text{ is not defined}) \end{aligned}$$

Each member of the set of programs returned by Ψ is a candidate for the body of the method to be evaluated. The set includes all methods provided by the ancestors of the current class that are not redefined. There are three possibilities.

1. If $|\Gamma| = 0$, there is no suitable method and evaluation fails.
2. If $|\Gamma| = 1$, there is exactly one applicable method, which is evaluated.
3. If $|\Gamma| > 1$, there are several applicable methods and the semantics must either forbid this situation or provide a strategy for choosing one of the methods.

6 Properties of the Model

The model has a number of useful properties, which are summarized in the following lemma. The proof of this lemma is straightforward, from the definitions we have given, but rather tedious.

Lemma Let P be a program and let $(V_1, E_1, c_1, a_1, r_1) = [\![P]\!](V_0, E_0, c_0, a_0, r_0)$. Then:

1. $V_1 \supseteq V_0$ and $c_1 = c_0$ and $a_1 = a_0$.
2. If P does not contain an assignment, then $E_1 = E_0$.

The model provides encapsulation. The value of an instance variable x can be accessed (by the program x) or changed (by the program $\mathsf{store}\ x$) only when its owner is the current object. Variables can be accessed from outside classes only if appropriate methods are provided, as in Smalltalk.

We could have defined, alternatively:

$$[\![x]\!](V, E, c, a, r) \quad = \quad (V, E, c, a, E(r, x)),$$

obtaining the instance variable from the result vertex rather than the current vertex. With this definition, we would have to write $(\mathsf{self}; x)$ to obtain the value of an instance variable of the current object, but we could access instance variables of other objects using extended sequences such as $(\mathsf{self}; x; y; \ldots)$.

All messages have exactly one argument and all methods have exactly one parameter, referenced as arg. This is not an inherent limitation of the formalism, because we can construct objects of arbitrary complexity to pass as arguments, but it does make the language tedious to use in practice. The advantage of the single argument is, of course, a considerable simplification of the semantic equations, because there is no need for parameter names or binding rules.

The model describes message passing at a relatively high level of abstraction. We could describe the machinery of message passing in the model. For example, we could introduce a linked list of vertices representing the run-time stack into the graph. The purpose of the semantics we have given, however, is to simplify reasoning by avoiding such clutter.

The usefulness of the model lies in its ability to describe both strong and weak features of object oriented programming.

A program has *side-effects* if there are states s for which evaluating it yields new edges in the state. Consequently, a program has side-effects only if it contains assignments. Note that the introduction of a new vertex is *not* a side-effect, because it is undetectable: it is the introduction of an edge to the new vertex that is the observable side-effect.

General purpose languages, such as Ada and C++, allow the programmer to choose between values (the objects themselves) and references (pointers to objects). Modern high level languages such as CLU, Eiffel, SML, and Smalltalk, however, provide references only. A compiler may use values for efficiency. Eiffel, for example, allows programmers to use values explicitly in the form of "expanded" objects [8]. The model provides a "reference semantics" in that edges in the graph correspond to references. Each object has a unique "identity", represented by a vertex in the graph.

The downside of identity, of course, is aliasing. Every vertex with in-degree greater than one corresponds to an alias [7]. Aliasing complicates reasoning about programs but, since it is a characteristic of many object oriented languages, it is appropriate that the model should describe it.

We use classes in the model only to define Φ. If we define Φ to depend on objects, rather than classes (that is, $\Phi: \mathcal{V} \to (\Sigma \to \Sigma)$), then we can model delegation.

7 Conclusion

Our intention was to model what we perceived to be the essential features of the object oriented paradigm but not to capture every feature of every language. The model that we have presented provides a simple interpretation of most of the features of most object oriented languages. We can describe any object oriented computation in the formal language. Non-trivial programs written in the formal language are rather obscure, but then so are non-trivial functions written as λ-terms. We can use the formal language to describe dynamic binding, object identity, side effects, aliasing, and other basic properties of object oriented programs.

There are some features of object oriented languages that the model cannot describe. Smalltalk, for instance, has higher order constructs—blocks and metaclasses—that cannot be expressed directly, although we can provide clumsy simulations of them. Although higher order constructs have been advocated for object oriented languages [2], they have not been widely adopted, which suggests that programmers may not be comfortable with them.

The model cannot represent "nested" objects—objects that are contained in other objects. Nesting serves two purposes: efficiency and hiding. Efficiency should not be a concern for a formalism, and we can model hiding, if necessary, by other means.

We have implemented the model, for experimental purposes, both as a functional program in SML and an object oriented program in Dee [5].

Acknowledgement The research described in this paper was funded by the Natural Sciences and Engineering Research Council of Canada. Patrice Chalin provided many helpful suggestions.

References

[1] P. America, J. de Bakker, J. Kok, and J. Rutten. Operational semantics of a parallel object-oriented language. In *Proc. Thirteenth ACM Symp. on Principles of Programming Languages*, pages 194–208, 1986.

[2] Henry G. Baker. Iterators: signs of weakness in object-oriented languages. *OOPS Messenger*, 4(3):18–25, July 1993.

[3] Kim B. Bruce. Safe type checking in a statically typed object-oriented programming language. In *Proc. Twentieth ACM Symp. on Principles of Programming Languages*, pages 285–298, January 1993.

[4] P. Canning, William Cook, W. Hill, W. Olthoff, and J. Mitchell. F-bounded polymorphism for object oriented programming. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, September 1989.

[5] Peter Grogono. Issues in the design of an object oriented programming language. *Structured Programming*, 12(1):1–15, January 1991.

[6] Peter Grogono and Mark Gargul. *A Refinement Calculus for Object Oriented Programming*. Technical Report OOP–93–1, Department of Computer Science, Concordia University, December 1993. 55 pages.

[7] John Hogg, Doug Lea, Alan Wills, Dennis de Champeaux, and Richard Holt. The Geneva convention on the treatment of object aliasing. *OOPS Messenger*, 3(2):11–16, April 1992. (Report on ECOOP'91 Workshop W3).

[8] Bertrand Meyer. *Eiffel: the Language*. Prentice Hall International, 1992.

[9] John C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *Proc. Seventeenth ACM Symp. on Principles of Programming Languages*, pages 109–124, January 1990.

[10] Benjamin C. Pierce and David N. Turner. Object-oriented programming without recursive types. In *Proc. Twentieth ACM Symp. on Principles of Programming Languages*, pages 299–312, January 1993.

[11] Alan Snyder. Modeling the C++ object model: an application of an abstract object model. In *European Conference on Object Oriented Programming*, pages 1–20, Springer-Verlag, 1991.