

A Computational Model for Object Oriented Programming

Peter Grogono

Mark Gargul

Department of Computer Science, Concordia University
Montréal, Québec H3G 1M8

Abstract

We describe a computational model for object oriented programs. A state of a computation is a directed graph in which each vertex represents an object and each edge represents an instance variable. Edges are labeled with instance variable names. We introduce a simple object oriented language and define a semantics which gives the meaning of each program as a function from states to states. Characteristic features of object oriented programming, such as polymorphism, inheritance, and dynamic binding, can be incorporated into the model in a natural way. We describe a simple type system. Programs are embedded in a complete lattice of specifications, allowing for the development of a refinement calculus.

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. 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 programming? 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 conventional approach to semantics is to describe simple types—booleans, integers, and so on—first, then introduce additional machinery for product, unions, and recursive types. Our approach uses a state that is more complicated than a simple environment, but which does not need to be extended for additional data structures.

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 another object. 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. In our model, edges are labeled with the names of instance variables.

Within this framework, we can model simple values, records, and recursive data structures 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.

Programming languages, being designed for mechanical translation, are necessarily formal. For some time, formal specification languages have also existed. But there are few examples of formalized translation from specifications to programs and, consequently, there is a gap in the software development process. There are good reasons for this gap: the transition from specification to program requires all the skills of programming. Nevertheless, there has been many proposals for *wide-spectrum languages* and—a more recent term for a similar idea—*refinement calculi*.

Hoare is perhaps the strongest advocate of another closely related idea: that specifications and programs should satisfy algebraic laws. Since functional and logic languages were designed in such a way that programs are respectable mathematical objects, Hoare's attention is directed towards imperative languages, which have a reputation for unruly behaviour. The usual way of taming imperative programs is to model them as functions on a set of states. Non-deterministic programs and specifications are modelled as relations on the same set of states. A *state* is a partial function mapping variable names to values.

Since object oriented programming is a variant of imperative programming, it is possible to adapt the model described above so that it handles objects. We believe that such adaptations are clumsy at best and, in this paper, we describe a formalism that captures the idiosyncrasies of the object oriented paradigm in a simple and useful way.

Imperative languages typically provide some primitive data types, such as *Bool* and *Int*, and a variety

of data structures, such as arrays, records, unions, and pointer structures. Refinement calculi typically emphasize simple variables and treat data structures as second-class citizens, if at all.

The object oriented paradigm focusses on objects rather than variables. Objects may be simple (e.g. integers and booleans) or compound (e.g. stacks and trees). Compound objects correspond to records. Unions are represented by instances of a subclass. Pointer structures are represented by objects that contain references to other objects. Arrays do not fit into the object oriented model quite so directly, but we can view an array as an object with indexed components. Thus a model for object oriented programming must be somewhat more complicated than a model for simple variables, but such a model unifies several concepts that are introduced rather arbitrarily into the simpler models.

3 The Graph Model

We model the state of an object oriented computation as a labeled, directed graph. Each vertex of the graph represents an object and each edge represents the value of an instance variable. Edges are labeled with the names of instance variables. Primitive values, such as booleans and integers, are represented by vertices with out-degree zero. During the evaluation of a message, the method invoked depends on both the value of the receiver (which is a vertex) and the name of the method.

Figure 1 shows the sets that we use in the model. 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	C	$Int, Bool, Root, \alpha, \beta, \gamma, \dots$
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}	n

Figure 1: Sets used in the model

A vertex has two components: a class name and an index chosen from the integers. Thus $\mathcal{V} = C \times \mathcal{Z}$. If $v = (\alpha, n)$ is a vertex, we write $v: \alpha$ (“ c is an instance of α ”). We also use the projection $\text{fst}(v)$ to select the class of v .

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.

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

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 $(\alpha, 1), (\alpha, 2), \dots$

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

$V \subseteq \mathcal{V}$	is a set of vertices,
$E: V \times \mathcal{L} \rightarrow V$	is the <i>edge</i> function,
$c \in V$	is the <i>current</i> vertex,
$a \in V$	is the <i>argument</i> vertex, and
$r \in V$	is the <i>result</i> vertex.

The partial function E defines the edges of the graph. If there is an edge from u to v with label x , the $E(u, x) = v$. Otherwise, E is undefined.

We occasionally use a dot notation for components of a state. For example, $s.V$ abbreviates “ V where $s = (V, E, c, a, r)$ ”. We use the symbol “ \dagger ” to define modified edge functions. If $E' = E \dagger (v, y, w)$ then

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

The state (V, E, c, a, r) models the state of an object oriented computation in the following way. Each vertex represents an object. If the object represented by a vertex u has an instance variable x whose value is the object represented by v , then $E(u, x) = v$. There are three distinguished vertices in the state. The vertex c represents the “current” object, variously called *current*, *self*, or *this* in practical languages; the vertex a represents the argument of the method currently being evaluated; and the vertex r represents the result of the previous operation.

We use 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, $!$, which may also be regarded as a *non-terminating* state. The set of all states is $\Sigma = \Sigma_t \cup \{!\}$.

3.1 Syntax

Figure 2 shows the abstract syntax of the language of our model. The first line shows *simple* programs. The programs in the second row, which contain other programs as proper components, are *compound* programs. The set \mathcal{P} consists of all programs generated by recursive applications of these productions. We use parentheses to disambiguate compound programs when necessary.

$$\begin{aligned} P &\rightarrow \text{skip} \mid \text{loop} \mid \text{true} \mid \text{false} \mid n \mid \text{self} \mid \text{arg} \mid \text{new } \alpha \mid x \mid \text{store } x \mid \\ P &\rightarrow P; Q \mid P \text{ else } Q \mid \text{while } P \text{ do } Q \mid m(Q) \end{aligned}$$

Figure 2: Syntax

3.2 Semantics

A common approach to defining the semantics of an imperative language is to separate programs into “expressions”, which have no side-effects, and “commands”, which have no value. The drawback with this approach is that it is awkward to describe a construct that both modifies the state and returns a value. In most practical languages, however, a “function” is just a procedure that happens to return a value. Our semantics is general: a program may affect the state and may yield a result.

Algol68 and, to a lesser extent, C are imperative languages that use this convention. Smalltalk is similar in that every message yields a result, the default result being the receiver of the message.

If $P \in \mathcal{P}$, then the denotation of P is a total function $\llbracket P \rrbracket: \Sigma \rightarrow \Sigma$. Figures 3 and 4 show the semantics of simple and compound programs respectively. Figure 5 gives the conditions under which the semantic equations hold. (It would be possible to include the validity conditions in the equations, but only with some loss of clarity.) If the conditions are not satisfied, the denotation of the specification is defined to be $\{!\}$. Also, $\llbracket P \rrbracket! = !$.

$$\begin{aligned}
\llbracket \text{skip} \rrbracket(V, E, c, a, r) &= (V, E, c, a, r) \\
\llbracket \text{loop} \rrbracket(V, E, c, a, r) &= ! \\
\llbracket \text{true} \rrbracket(V, E, c, a, r) &= (V \cup \{\mathbf{v}_{\text{true}}\}, E, c, a, \mathbf{v}_{\text{true}}) \\
\llbracket \text{false} \rrbracket(V, E, c, a, r) &= (V \cup \{\mathbf{v}_{\text{false}}\}, E, c, a, \mathbf{v}_{\text{false}}) \\
\llbracket n \rrbracket(V, E, c, a, r) &= (V \cup \{\mathbf{v}_n\}, E, c, a, \mathbf{v}_n) \\
\llbracket \text{self} \rrbracket(V, E, c, a, r) &= (V, E, c, a, c) \\
\llbracket \text{arg} \rrbracket(V, E, c, a, r) &= (V, E, c, a, a) \\
\llbracket \text{new } \alpha \rrbracket(V, E, c, a, r) &= (V \cup \{\mathbf{v}\}, E, c, a, \mathbf{v}) \\
&\quad \text{where } \mathbf{v} = (\alpha, 1 + \max \{n \mid (\alpha, n) \in V\}) \\
\llbracket x \rrbracket(V, E, c, a, r) &= \{(V, E, c, a, E(c, x))\} \\
\llbracket \text{store } x \rrbracket(V, E, c, a, r) &= \{(V, E \dagger (c, x, r), c, a, r)\}
\end{aligned}$$

Figure 3: Semantics 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. Strictly, the program “`n`” is a numeral; its denotation contains (Int, n) , in which n is an integer.

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 result is $!$. The program `store x` is the inverse of x : it yields a state in which $E(c, x) = r$.

The program `new α` introduces a new vertex into the graph. The new vertex is (α, n) , in which n is a unique index for the class α . In Figure 3, n is one greater than the maximum index of the class. We do not use `new` with the primitive classes `Bool` or `Int` because instances of these classes are introduced by literals.

Sequences are defined by functional composition. The conditional statement `P else Q` is reminiscent of Smalltalk. 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}_{true} or $\mathbf{v}_{\text{false}}$, or some other value.

We define `while P do Q` as the fixed point of a recursive equation. We discuss the choice of a suitable fixed point in Section 7.3.

The program `$m(Q)$` sends a “message” to the result object. In most object oriented languages, the syntax of a message includes the receiver. We can interpret the denotation of `$m(Q)$` operationally

$$\begin{aligned}
\llbracket P; Q \rrbracket s &= \llbracket Q \rrbracket (\llbracket P \rrbracket s) \\
\llbracket P \text{ else } Q \rrbracket s &= \begin{cases} \llbracket P \rrbracket s, & \text{if } s.r = \mathbf{v}_{\text{true}}, \\ \llbracket Q \rrbracket s, & \text{if } s.r = \mathbf{v}_{\text{false}}, \\ !, & \text{otherwise.} \end{cases} \\
\llbracket \text{while } P \text{ do } Q \rrbracket s &= \llbracket \mu X. (P; ((Q; X) \text{ else skip})) \rrbracket s \\
\llbracket m(Q) \rrbracket (V_0, E_0, c_0, a_0, r_0) &= (V_2, E_2, c_0, a_0, r_2) \\
&\text{where } (V_1, E_1, c_1, a_1, r_1) = \llbracket Q \rrbracket (V_0, E_0, c_0, a_0, r_0) \\
&\text{and } (V_2, E_2, c_2, a_2, r_2) = \llbracket \Phi(\text{fst}(r_0), m) \rrbracket (V_1, E_1, r_0, r_1, r_1)
\end{aligned}$$

Figure 4: Semantics of Compound Programs

Specification	is well defined iff:
$\llbracket x \rrbracket (V, E, c, a, r)$	$E(c, x)$ is defined
$\llbracket P \text{ else } Q \rrbracket (V, E, c, a, r)$	$r \in \{\mathbf{v}_{\text{true}}, \mathbf{v}_{\text{false}}\}$
$\llbracket \text{new } \alpha \rrbracket (V, E, c, a, r)$	$\alpha \notin \{\text{Bool}, \text{Int}\}$
$\llbracket m(Q) \rrbracket (V, E, c, a, r)$	$\llbracket Q \rrbracket (V, E, c, a, r) \neq !$ and $\llbracket \Phi(\text{fst}(r_0), m) \rrbracket (V_1, E_1, r_0, r_1, r_1) \neq !$

Figure 5: Validity Conditions

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 Q 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 value of $\Phi(\text{fst}(r_0), m)$ is the body of the method with name m in class $\text{fst}(r_0)$, the class of the receiver. The body is evaluated in the state $(V_1, E_1, r_0, r_1, r_1)$, in which the current object is the receiver and the argument vertex is the value of Q . Evaluating the body yields the new state $(V_2, E_2, c_2, a_2, r_2)$.
4. The final state is $(V_2, E_2, c_0, a_0, r_2)$. The values of the current object and the argument are restored from the initial state.

The partial function Φ takes a class name and a method name as arguments and returns a program. There are two possibilities: if the method is “user defined”, the value returned is in \mathcal{P} . The primitive classes *Bool* and *Int* contain primitive methods that cannot be expressed easily in the language. For these methods, the value returned is a function on states. Thus:

$$\Phi: \mathcal{V} \times \mathcal{M} \rightarrow \mathcal{P} \cup (\Sigma \rightarrow \Sigma).$$

Consider, for example, the message $(2; \text{add}(3))$. Evaluation requires $\Phi(\text{Int}, \text{add})$, which is a primitive method of the class *Int*. This method, add_{Int} , is defined by

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

if $s.c$ and $s.a$ are instances of class *Int*, and ! otherwise. Alternatively, Φ might return a user-defined function such as

$$\Phi(\text{Int}, \text{avg}) = (\text{self}; \text{add}(\text{arg}); \text{div}(2))$$

in which case the message $(4; \text{avg}(6))$ would lead to

$$\begin{aligned} \Phi(\text{Int}, \text{avg})(V, E, \mathbf{v}_4, \mathbf{v}_6, r) &= \llbracket \text{self}; \text{add}(\text{arg}); \text{div}(2) \rrbracket(V, E, \mathbf{v}_4, \mathbf{v}_6, r) \\ &= \llbracket \text{add}(\text{arg}); \text{div}(2) \rrbracket(V, E, \mathbf{v}_4, \mathbf{v}_6, \mathbf{v}_4) \\ &= \llbracket \text{div}(2) \rrbracket(V, E, \mathbf{v}_4, \mathbf{v}_6, \mathbf{v}_{10}) \\ &= (V, E, \mathbf{v}_4, \mathbf{v}_6, \mathbf{v}_5) \end{aligned}$$

The definitions of Figure 3 are sufficient for theoretical studies. To complete the model, however, we show how to evaluate a complete program from an initial state. We assume a value for Φ that provides user-defined and standard methods for appropriate arguments. In particular, $\Phi(\text{Root}, \text{main})$ is the “main program”. The result of evaluating the program is the state

$$\llbracket \mathbf{v}; \text{main}(\mathbf{v}) \rrbracket(\{\mathbf{v}\}, \{\}, \mathbf{v}, \mathbf{v}, \mathbf{v})$$

where $\mathbf{v} = (\text{Root}, 1)$. The only occurrence of \mathbf{v} that is actually used in the evaluation is the first one. The others are merely placeholders ensuring that each component of the initial state is well-defined.

Several kinds of program depend on the result of the previous computation. The programs will normally be used as the second component of a sequence. We can improve the readability of the language by defining the “macros” such as the following.

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

4 Examples

In the formal language, all methods require exactly one argument. Many of the methods introduced in this section are unary functions that do not access their argument. Other methods are binary functions that do access their arguments. We adopt the convention that, in a function definition, the name of a binary function is followed by “()” to indicate that an argument is expected.

4.1 Booleans

A Boolean object is an instance of either class *True* or class *False*. Each class provides three methods, *not*, *and*, and *or*, with different implementations. Figure 6 shows these classes. The table provides elements of the function Φ . The formal expression of Figure 6, for example, would be

$$\Phi = \{ (True, not, (new False)), (True, and, (arg)), (True, or, (new True)), \dots, \\ (False, not, (new True)), (False, and, (new False)), (False, or, (arg)), \dots \}$$

Method	class <i>True</i>	class <i>False</i>
<i>not</i>	<i>new False</i>	<i>new True</i>
<i>and</i> ()	<i>arg</i>	<i>new False</i>
<i>or</i> ()	<i>new True</i>	<i>arg</i>
<i>implies</i> ()	<i>arg</i>	<i>new True</i>
<i>iff</i> ()	<i>arg</i>	<i>arg; not</i>

Figure 6: The Classes *True* and *False*

4.2 Natural Numbers

Instances of the class *Zero* represent the number 0; instances of the class *Pos* represent numbers greater than zero. Figure 7 shows these classes. The blank entries in class *Zero* indicate that this class does not provide the methods *link* and *pred*.

The natural number $n > 0$ is represented by an instance u_1 of class *Pos*, where u_1 is the root of a linked list containing $n - 1$ edges. In other words, the state graph contains the edges

$$(u_1, next, u_2), \dots, (u_{n-1}, next, u_n).$$

Method	class <i>Zero</i>	class <i>Pos</i>
<i>iszero</i>	new <i>True</i>	new <i>False</i>
<i>succ</i>	new <i>Pos</i> ; store <i>x</i> ; <i>link</i> (self); <i>x</i>	new <i>Pos</i> ; store <i>x</i> ; <i>link</i> (self); <i>x</i>
<i>link</i> ()		arg; store <i>next</i>
<i>pred</i>		<i>next</i>
<i>add</i> ()	arg	<i>pred</i> ; <i>add</i> (arg); <i>succ</i>
<i>eq</i> ()	arg; <i>iszero</i>	arg; <i>eqr</i> (self)
<i>eqr</i> ()	new; <i>False</i>	arg; <i>pred</i> ; <i>eq</i> (self; <i>Pred</i>)
<i>lt</i> ()	arg; <i>iszero</i> ; <i>not</i>	arg; <i>ltr</i> (self)
<i>ltr</i> ()	new <i>False</i>	arg; <i>pred</i> ; <i>lt</i> (self; <i>pred</i>)

Figure 7: The Classes *Zero* and *Pos*

The call $v; succ()$, in which v is a vertex representing the natural number n , constructs a vertex representing $n + 1$ by creating a new node and linking it to n via an edge *next*, using the auxiliary method *link*. The edge label *next* must be the same for all numbers, but is not a reserved name — it can be freely used for other purposes.

The call $v; pred()$, in which v is a vertex representing the natural number n , returns the tail of the list rooted at v , which represents the predecessor of n .

Some functions, such as *eq* (equal) and *lt* (less than) require auxiliary functions (*eqr* and *ltr*). The auxiliary functions perform a similar operation to the primary functions, but with the argument and receiver interchanged. This corresponds to “double dispatching”.

A consequence of the use of double dispatching is that the classes *True* and *False* do not play an important role in the implementation of natural numbers. Only the predicates *iszero*, *eq*, and *lt* return instances of *True* and *False*. This is because we have taken to an extreme the idea that object oriented programs should handle case analysis by dynamic binding.

4.3 A Program

We adopt the convention that a “main program” is started by invoking the method *entry* in the class *Main*. Here is a simple program:

```
class Main
  method entry
    new Zero; succ; store one;
    succ; add(self; one)
```

This program computes the successor of zero, stores it as “*one*”, computes the successor of the same number, and adds “*one*”. The result is a data structure that represents the natural number 3. Figures 8 and 9 show the computation in tabular form.

The first column shows the statement executed. The next three columns show the effect of the statement on the special vertices c , a , and r . The fifth column shows new edges added to the graph. The columns contain only entries that are relevant to the computation; a blank indicates that nothing has happened. The numbers in the sixth column correspond to the notes below.

Program	<i>c</i>	<i>a</i>	<i>r</i>	<i>E</i>	Note
<i>new Zero</i>	r_1		z_1		1
<i>succ</i>	z_1				
<i>new Pos</i>			p_1		
store <i>x</i>				(z_1, x, p_1)	2
<i>link(self)</i>	p_1	z_1			
arg			z_1		
store <i>next</i>				$(p_1, next, z_1)$	
ret	z_1				3
<i>x</i>			p_1		
ret	r_1				
store <i>one</i>				(r_1, one, p_1)	
<i>succ</i>	p_1				
<i>new Pos</i>			p_2		
store <i>x</i>				(p_1, x, p_2)	
<i>link(self)</i>	p_2	p_1			
arg			p_1		
store <i>next</i>				$(p_2, next, p_1)$	
ret	p_1				
<i>x</i>			p_2		
ret	r_1				

Figure 8: Execution of program: first part

Program	<i>c</i>	<i>a</i>	<i>r</i>	<i>E</i>	Note
<i>add(self; one)</i>	p_2	p_1			4
<i>self</i>			p_2		
<i>pred</i>	p_2				
<i>next</i>			p_1		
<i>ret</i>	p_2				
<i>add(arg)</i>	p_1	p_1			
<i>self</i>			p_1		
<i>pred</i>	p_1				
<i>next</i>			z_1		
<i>ret</i>	p_1				
<i>add(arg)</i>	z_1	p_1			
<i>arg</i>			p_1		
<i>ret</i>	p_1				
<i>succ</i>	p_1				
<i>new Pos</i>			p_3		
<i>store x</i>				(p_1, x, p_3)	
<i>link(self)</i>	p_3	p_1			
<i>arg</i>			p_1		
<i>store next</i>				$(p_3, next, p_1)$	
<i>ret</i>	p_1				
<i>x</i>			p_3		
<i>ret</i>	p_1				
<i>ret</i>	p_2				
<i>succ</i>	p_3				
<i>new Pos</i>			p_4		
<i>store x</i>				(p_3, x, p_4)	
<i>link(self)</i>	p_4	p_3			
<i>arg</i>			p_3		
<i>store next</i>				$(p_4, next, p_3)$	
<i>ret</i>	p_3				
<i>x</i>			p_4		
<i>ret</i>	p_2				
<i>ret</i>	r_1				5

Figure 9: Execution of program: second part

1. The program is implicitly initialized by creating the vertex $(Root, 1)$, abbreviated to r_1 in the table. In similar fashion, z_i stands for $(Zero, i)$, and p_i stands for (Pos, i) .
2. The column headed E does not show the entire edge relation, but only the edge most recently added to the state graph. Although in general a new edge may override an old edge, this does not happen in the example.
3. The program `ret` is not part of the formal language. It is used in the table to show the restored context after a method has returned. Although both the current object (c) and the argument (a) are restored, the table shows only the current object.
4. The method `add` is invoked three times, illustrating recursion. The recursion terminates when the receiver of `add` is an instance of `Zero`.
5. When the program terminates, the current vertex is r_1 , as it was initially. The result is p_4 , which corresponds to the natural number 3, since

$$\{(p_4, next, p_3), (p_3, next, p_1), (p_1, next, z1) \subseteq E\}.$$

The example shows a rather “functional” style of programming. To see the difference between our model and a purely functional model, consider adding the following method to class `Pos`:

```
method infinity
  new Pos; store x; link(x)
```

Writing ∞ to stand for this object, and using mathematical notation, the model can execute the following programs in bounded time (n stands for any finite instance of `Pos`):

$$\begin{aligned} succ(\infty) &\rightarrow \infty \\ pred(\infty) &\rightarrow \infty \\ n < \infty &\rightarrow True \\ n + \infty &\rightarrow \infty \end{aligned}$$

The following computations, however, do not terminate:

$$\begin{aligned} \infty + n \\ \infty + \infty \\ \infty = \infty \end{aligned}$$

5 Properties of the Model

The semantics gives a denotation for all syntactically well-formed specifications. It has a number of useful properties, which are summarized in Lemma 5.1. The word “contain” in this lemma is used in the following sense: a program P contains an entity Q if Q occurs either in the text of P or in the text of a method called by P .

Lemma 5.1 *Let P be a program and let $(V_1, E_1, c_1, a_1, r_1) = \llbracket P \rrbracket (V_0, E_0, c_0, a_0, r_0)$. Then:*

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

The semantics of Figure 3 provides encapsulation. The value of an instance variable can be accessed (x) or changed (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.

As an alternative, we could define

$$\llbracket x \rrbracket(V, E, c, a, r) = (V, E, c, a, E(r, x)),$$

obtaining the instance variable from the result vertex rather than the current vertex. With this rule, we have to write (self; x) to obtain the value of an instance variable of the current object, but we can access instance variables of other objects using extended sequences such as (self; x ; y ; . . .).

All messages have exactly one argument and all methods have exactly one parameter, referenced as *arg*. This is not an inherent limitation of the language, 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 and binding rules.

The semantic equations describe message passing at a relatively high level of abstraction. We could describe the machinery of message passing by introducing a linked list of vertices representing the stack into the graph. The purpose of the semantics we have given, however, is to simplify reasoning by avoiding such clutter.

The properties enable us to give a simple characterization of side-effects. A program P has side-effects if there are states s for which $(\llbracket P \rrbracket s).E \neq s.E$. A program has side-effects iff it contains assignments.

The semantics is of the kind usually called “reference semantics” because 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—which the model also describes.

The semantics describes aliasing because vertices with in-degree greater than one can be constructed. Aliasing complicates reasoning about programs but, since it is a characteristic of many object oriented languages, it is appropriate that the semantics should describe it correctly.

The semantics is class-based: the methods provided by an object depend on its class. It would be possible to model delegation by allowing each object to provide its own set of methods.

5.1 Inheritance

Proponents of the object oriented paradigm will point out that our model does not mention inheritance. It is not difficult to include inheritance, however, and we propose one method here.

We define a partial order, \leq , on class names. If $\alpha \leq \beta$, we say “class α inherits from class β ”. We introduce a function Φ_{\leq} which generalizes Φ :

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

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

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

Each member of the set of programs returned by Φ_{\preceq} is a candidate for the method to be evaluated. The set includes all methods provided by the ancestors of the current class that are not overridden. There are three possibilities.

- If $|\Phi_{\preceq}(c, m)| = 0$, there is no suitable method and evaluation fails.
- If $|\Phi_{\preceq}(c, m)| = 1$, there is exactly one applicable method, which is evaluated.
- If $|\Phi_{\preceq}(c, m)| > 1$, there are several applicable methods and the semantics must either forbid this situation or provide a strategy for choosing one of the methods.

5.2 Type Checking

The original purpose of the graph model was to reason about the correctness of object oriented programs without executing them. Verification and execution are the extremes of a spectrum of computations that we can perform on the text of a program. Type checking falls between these extremes; it is a useful compromise because it can be performed in bounded time and it yields useful information.

The type system that we describe in this section is too simple to be of practical use on object oriented programming. It is intended as a demonstration of the kind of reasoning that the model accommodates.

We define the partial function $\Psi: C \times \mathcal{L} \rightarrow C$ as follows: $\Psi(\alpha, x) = \beta$ means that the value of the instance variable x of class α is required to be an instance of class β . The functions Φ and Ψ play the role of method declarations and instance variable declarations, respectively. We could introduce them into the language, but there does not appear to be any particular advantage in doing so.

A state $s = (V, E, c, a, r)$ is *well-typed with respect to Ψ* , written $\mathcal{W}(s)$, if

$$\forall u, \alpha, x. u: \alpha \wedge E(u, x) = v \Rightarrow v: \Psi(\alpha, x).$$

If P is a well-behaved program, we write $P: \text{comm}$. A stronger condition is that P is well-behaved and yields an instance of class α , written as $P: \text{class}(\alpha)$.

We write (κ, α, ρ) for the assumptions $c: \kappa$, $a: \alpha$, and $r: \rho$ in the current state. The assertion $(\kappa, \alpha, \rho) \vdash P: \text{comm}$ means: “ P is a well-behaved in every well-typed state in which $c: \kappa$, $a: \alpha$, and $r: \rho$ ”. The assertion $(\kappa, \alpha, \rho) \vdash P: \text{class}(\beta)$ means: “ P yields an instance of β in every well-typed state in which $c: \kappa$, $a: \alpha$, and $r: \rho$ ”. When the class of a vertex is irrelevant, we replace the class name by an underscore. For example, $(_, \alpha, _)$ is the assumption that the class of the argument is α .

Figures 10 and 11 show axioms and inference rules for good behavior. The first seven axioms and the axiom NEW apply to programs that never fail. Axiom IVAR requires the appropriate edge to exist. Rule STORE requires the result vertex to be a member of the appropriate class for the new edge.

[SKIP]	$(_, _, _) \vdash \text{skip: comm}$
[LOOP]	$(_, _, _) \vdash \text{loop: comm}$
[TRUE]	$(_, _, _) \vdash \text{true: class}(Bool)$
[FALSE]	$(_, _, _) \vdash \text{false: class}(Bool)$
[NUM]	$(_, _, _) \vdash n: \text{class}(Int)$
[SELF]	$(\kappa, _, _) \vdash \text{self: class}(\kappa)$
[ARG]	$(_, \alpha, _) \vdash \text{arg: class}(\alpha)$
[IVAR]	$(\kappa, _, _) \vdash x: \text{class}(\Psi(\kappa, x))$ if $\Psi(\kappa, x)$ is defined
[STORE]	$(\kappa, _, \Psi(\kappa, x)) \vdash \text{store } x: \text{comm}$ if $\Psi(\kappa, x)$ is defined
[NEW]	$(_, _, _) \vdash \text{new } \beta: \text{class}(\beta)$

Figure 10: Well-behavedness rules

[COMM]	$\frac{(\kappa, \alpha, \rho) \vdash P: \text{class}(\beta)}{(\kappa, \alpha, \rho) \vdash P: \text{comm}}$
[SEQ ₁]	$\frac{(\kappa, \alpha, \rho) \vdash P: \text{class}(\beta) \quad (\kappa, \alpha, \beta) \vdash Q: \text{class}(\gamma)}{(\kappa, \alpha, \rho) \vdash P; Q: \text{class}(\gamma)}$
[SEQ ₂]	$\frac{(\kappa, \alpha, \rho) \vdash P: \text{class}(\beta) \quad (\kappa, \alpha, \beta) \vdash Q: \text{comm}}{(\kappa, \alpha, \rho) \vdash P; Q: \text{comm}}$
[COND ₁]	$\frac{(\kappa, \alpha, \rho) \vdash P: \text{class}(\beta) \quad (\kappa, \alpha, \rho) \vdash Q: \text{class}(\beta)}{(\kappa, \alpha, \rho) \vdash P \text{ else } Q: \text{class}(\beta)}$
[COND ₂]	$\frac{(\kappa, \alpha, \rho) \vdash P: \text{comm} \quad (\kappa, \alpha, \rho) \vdash Q: \text{comm}}{(\kappa, \alpha, \rho) \vdash P \text{ else } Q: \text{comm}}$
[WHILE]	$\frac{(\kappa, \alpha, \rho) \vdash P: \text{class}(Bool) \quad (\kappa, \alpha, \rho) \vdash Q: \text{comm}}{(\kappa, \alpha, \rho) \vdash \text{while } P \text{ do } Q: \text{comm}}$
[MESS ₁]	$\frac{(\kappa, \alpha, \rho) \vdash Q: \text{class}(\beta) \quad (\rho, \beta, _) \vdash \Phi(\rho, m): \text{class}(\gamma)}{(\kappa, \alpha, \rho) \vdash m(Q): \text{class}(\gamma)}$ if $\Phi(\rho, m)$ is defined
[MESS ₂]	$\frac{(\kappa, \alpha, \rho) \vdash Q: \text{class}(\beta) \quad (\rho, \beta, _) \vdash \Phi(\rho, m): \text{comm}}{(\kappa, \alpha, \rho) \vdash m(Q): \text{comm}}$ if $\Phi(\rho, m)$ is defined

Figure 11: Well-behavedness rules

Rule COMM allows us to discard the result of an “expression”, thereby turning the expression into a “command”. The two sequence rules provide for sequences of expressions and sequences of commands, respectively. The two conditional rules and the two message rules are similar. The message rules ensure that the method exists (Φ is defined) and that the receiver and argument types are appropriate.

Lemma 5.2

1. If $P:\text{comm}$ can be inferred from the rules of Figures 10 and 11, then

$$\forall s \in \Sigma_t . \mathcal{W}(s) \Rightarrow \mathcal{W}(\llbracket P \rrbracket s).$$

2. If $P:\text{class}(\alpha)$ can be inferred from the rules of Figures 10 and 11, then

$$\begin{aligned} \forall s \in \Sigma_t . \mathcal{W}(s) \Rightarrow \mathcal{W}(s') \wedge s'.r:\alpha \\ \text{where } s' = \llbracket P \rrbracket s. \end{aligned}$$

We prove Lemma 5.2 by showing that each of the axioms and inference rules are valid with the definitions we have given.

Figure 12 uses the type to establish that the program $(\text{self}; \text{add}(\text{arg}); \text{div}(2))$ yields an integer if it is evaluated with an integer as the current vertex and an integer as the argument. The justification *Int* indicates that the rule used is a property of the class *Int*.

1	$(\text{Int}, _, _) \vdash \text{self}:\text{class}(\text{Int})$	SELF
2	$(_, \text{Int}, _) \vdash \text{arg}:\text{class}(\text{Int})$	ARG
3	$(\text{Int}, \text{Int}, _) \vdash \text{add}:\text{class}(\text{Int})$	<i>Int</i>
4	$(\text{Int}, \text{Int}, _) \vdash \text{add}(\text{arg}):\text{class}(\text{Int})$	MESS ₁ , 2, 3
5	$(\text{Int}, \text{Int}, _) \vdash \text{self}; \text{add}(\text{arg}):\text{class}(\text{Int})$	SEQ ₁ , 1, 4
6	$(_, _, _) \vdash 2:\text{class}(\text{Int})$	<i>Int</i>
7	$(\text{Int}, \text{Int}, _) \vdash \text{div}:\text{class}(\text{Int})$	<i>Int</i>
8	$(\text{Int}, \text{Int}, _) \vdash \text{div}(2):\text{class}(\text{Int})$	MESS ₁ , 6, 7
9	$(\text{Int}, \text{Int}, _) \vdash \text{self}; \text{add}(\text{arg}); \text{div}(2):\text{class}(\text{Int})$	SEQ ₁ , 5, 8

Figure 12: Type checking $(\text{self}; \text{add}(\text{arg}); \text{div}(2))$

Inheritance makes type-checking harder . . .

6 Program Development by Refinement

We develop programs by working from an abstract specification towards a concrete implementation.

7 Specifications

A relation $R \subseteq \Sigma \times \Sigma$ is a *specification* provided that $(!,!) \in R$, and $(!,s) \notin R$ for any $s \neq !$. (A more concise way of expressing this condition: $(!,s) \in R$ iff $s = !$.)

If R is a relation and i is a state then the *image* Ri of i under R is the set

$$Ri = \{ o \mid (i, o) \in R \}.$$

A specification is *total* iff, for all states i , we have $|Ri| \geq 1$. A specification is *deterministic* iff, for all states i , we have $|Ri| \leq 1$. A *program* is a total and deterministic specification or, in other words, a function on Σ . We can justify the requirement that programs be total in two ways. Intuitively, the automaton is either in the process of evaluating a program or it is in some definite state. When we start it, it should run for a while and then stop. If it does not stop, or blows up, its state is not observable, which we have denoted as $!$. Mathematically, it is easier to work with total functions than with partial functions.

The requirement that programs be deterministic is not crucial to the theory. All of the computations we define in the paper happen to be deterministic. It would not be difficult to add a non-deterministic choice operator to the language, but this would introduce the risk of paradoxes without greatly increasing the expressiveness of the programming language. During refinement, we may use non-deterministic *specifications*, but the final program is deterministic.

Specifications are ordered by the subset ordering, \subseteq . A specification or program R *satisfies* a specification R' iff $R \subseteq R'$. Intuitively, a specification allows a variety of behaviors. A program that satisfies the specification must provide one of the allowed behaviors. For example, if $Ri = \{o_1, o_2, o_3\}$, and $Pi = \{o_2\}$, then P satisfies R for the initial state i . Our model differs from others in that programs are either equal or incomparable.

The specification with greatest cardinality is fail:

$$\llbracket \text{fail} \rrbracket = (\Sigma_t \times \Sigma) \cup \{(!, !)\}.$$

We use brackets $\llbracket \cdot \rrbracket$ to distinguish the syntax, or name, of a specification from its denotation. fail is the “worst” specification because it allows every possible behavior. The specification with smallest cardinality is

$$\llbracket \text{miracle} \rrbracket = \{(!, !)\}.$$

This specification cannot be satisfied by any program. In general, a specification that is not total is not satisfiable.

Although the set of specifications is not a powerset, it is closed under union, intersection, and relational composition.

Lemma 7.1 *Specifications form a complete lattice, bounded by fail and miracle, with \cup and \cap as join and meet operations.*

Specification lattices are usually oriented with the strongest specification at the top and the weakest at the bottom, using an information ordering. In our case, the strongest specification, miracle, has the smallest cardinality, suggesting that it should be at the bottom of the lattice. Since we want to use the symbols \cup , \cap , and \subseteq of set theory, we place miracle at the *bottom* of the lattice, with fail at the top. To minimize confusion, we will use the names miracle and fail, rather than the symbols \top and \perp , for the extrema of the lattice.

The programs `skip` and `loop`, defined in Section 3.2 above, are also specifications. Another specification that we use below is `chance`, defined by

$$\text{chance} \equiv (\Sigma_t \times \Sigma_t) \cup \{(!, !)\}.$$

7.1 Predicates

If A is a predicate on the set of states, we write \overline{A} to denote the set of states in which A is true:

$$\overline{A} \equiv \{s \in \Sigma \mid A(s)\}.$$

We also define a specification, \widehat{A} , corresponding to a predicate A :

$$\widehat{A} \equiv \{(s, s) \mid s \in \overline{A}\} \cup \{(!, !)\}.$$

We note that

$$\begin{aligned} \widehat{\text{true}} &= \text{skip}, \text{ and} \\ \widehat{\text{false}} &= \text{miracle}. \end{aligned}$$

7.2 Prescriptions

Informally, a *prescription*, $[A \mid B]$ is a specification with precondition A and postcondition B . Formally, we give two definitions for prescriptions.

$$[A \mid B] = \overline{A} \times \overline{B} \cup \overline{\neg A} \times \Sigma \cup \{(!, !)\} \quad (1)$$

$$[A \mid B] = \text{chance} \circ \widehat{B} \cup \widehat{\neg A} \circ \text{fail} \quad (2)$$

Lemma 7.2 *Definitions (1) and (2) are equivalent.*

Definition (1) is direct and easy to understand. Definition (2) is less obvious, but has the advantage that it fits into the algebra of specifications.

Lemma 7.3 *If P is a program, $P \sqsubseteq [A \mid B]$, and $s \in \overline{A}$, then $\llbracket P \rrbracket s \in \overline{B}$.*

A program P is a *predicate program*, abbreviated “PP”, iff for all states (V, E, c, a, r) , we have $\llbracket P \rrbracket (V, E, c, a, r) = (V', E', c, a, r')$ and $r' \in \{\mathbf{v}_{\text{true}}, \mathbf{v}_{\text{false}}\}$, $V \sqsubseteq V'$, and $E = E'$.

Introduce the notation for prescriptions: $[A \mid B]$. Corresponding to each kind of program, there are many refinement rules. Figure 13 shows a simple set, not including messages.

- Binding strength of prescription operator.
- Simple programs have axioms and compound programs have inference rules.
- Clarify the relation between predicates and programs that (are supposed to) yield boolean values.

- Provide a rule for P else Q rather than if P then Q else R .
- Note the nice symmetry between x and store x !
- Loop rules requires termination.

$$\begin{aligned}
\text{skip} &\sqsubseteq [A | A] \\
\text{true} &\sqsubseteq [\text{true} | r = \mathbf{v}_{\text{true}}] \\
\text{false} &\sqsubseteq [\text{true} | r = \mathbf{v}_{\text{false}}] \\
n &\sqsubseteq [\text{true} | r = n] \\
\text{self} &\sqsubseteq [\text{true} | r = c] \\
\text{arg} &\sqsubseteq [\text{true} | r = a] \\
\text{new } \alpha &\sqsubseteq [\text{true} | r : \alpha] \\
x &\sqsubseteq [E(c, x) = e | r = e] \\
\text{store } x &\sqsubseteq [r = e | E(c, x) = e]
\end{aligned}$$

$$\begin{aligned}
&\frac{P \sqsubseteq [A | C] \quad Q \sqsubseteq [C | B]}{(P; Q) \sqsubseteq [A | B]} \\
&\frac{r = \mathbf{v}_{\text{true}} \quad P \sqsubseteq [A | B]}{P \text{ else } Q \sqsubseteq [A | B]} \\
&\frac{r = \mathbf{v}_{\text{false}} \quad Q \sqsubseteq [A | B]}{P \text{ else } Q \sqsubseteq [A | B]} \\
&\frac{Q \sqsubseteq [A \wedge P | A]}{(\text{while } P \text{ do } Q) \sqsubseteq [A | A \wedge \neg P]}
\end{aligned}$$

Figure 13: Refinement Rules

7.3 Loops

In Figure 4, we followed the usual practice of defining $\text{while } P \text{ do } Q$ as a fixed point of the function F defined by

$$F(X) = \text{if } P \text{ then } (Q; X) \text{ else skip.}$$

By Tarski's theorem, the sequences

$$\text{fail} = F^0(\text{fail}) \supseteq F^1(\text{fail}) \supseteq \dots \supseteq F^n(\text{fail}) \supseteq \dots$$

and

$$\text{miracle} = F^0(\text{miracle}) \sqsubseteq F^1(\text{miracle}) \sqsubseteq \dots \sqsubseteq F^n(\text{miracle}) \sqsubseteq \dots$$

have limits $\lim_{n \rightarrow \infty} F^n(\text{fail})$ and $\lim_{n \rightarrow \infty} F^n(\text{miracle})$ which are the extrema of a complete lattice of fixed points of F . It is customary to take the fixed point with least information which, in our case, is $\lim_{n \rightarrow \infty} F^n(\text{fail})$. This approach, however, does not fit comfortably into our model. If the loop terminates, it has the value that we expect: $(C; P)^k$ for some $k \geq 0$. If the loop does not terminate, $\lim_{n \rightarrow \infty} F^n(\text{fail})$ is not in general a program, because fail is not a program.

Consider, for example, the program `while true do Q`. We have

$$\begin{aligned} F(X) &= \text{if true then } (Q; X) \text{ else skip} \\ &= Q; X \end{aligned}$$

Since $Q; \text{fail} = \text{fail}$ for all programs Q , it is easy to show that $F^n(\text{fail}) = \text{fail}$ for all $n \geq 0$ and hence that $\lim_{n \rightarrow \infty} F^n(\text{fail}) = \text{fail}$. But this means that the program `while true do Q` may do anything at all. We would prefer a semantics which gave the value `loop` to this program.

One possibility would be to define the meaning of `while P do Q` to be $\lim_{n \rightarrow \infty} F^n(\text{loop})$, with F defined as before. Since $Q; \text{loop} = \text{loop}$ if Q is total, we have $F^n(\text{loop}) = \text{loop}$ for all $n \geq 0$, as required. But the limit $\lim_{n \rightarrow \infty} F^n(\text{loop})$ does not exist for all programs P and Q , so we cannot use it to define the meaning of iterative programs.

8 Conclusion

Make of it what you will.