# A Denotational Semantics of Inheritance and its Correctness

William Cook
Apple Computer
20525 Mariani Ave, 37 UP
Cupertino, CA 95014
WILLIAM@AppleLink.Apple.COM

Jens Palsberg
Computer Science Department
Aarhus University
Ny Munkegade
DK-8000 Aarhus C, Denmark
palsberg@daimi.aau.dk

**Abstract**

This paper presents a denotational model of inheritance. The model is based on an intuitive motivation of inheritance as a mechanism for deriving modified versions of recursive definitions. The correctness of the model is demonstrated by proving it equivalent to an operational semantics of inheritance based upon the method lookup algorithm of object-oriented languages.

*Information and Computation*, 114(2):329–350, 1994.
Also in Proc. Fourth ACM Conference on Object-Oriented Programming: Systems, Languages and Applications, New Orleans, October 1989, pages 433–443.

## 1 Introduction

Inheritance is one of the central concepts in object-oriented programming. Despite its importance, there seems to be a lack of consensus on the proper way to describe inheritance. This is evident from the following review of various formalizations of inheritance that have been proposed.

The concept of *prefixing* in Simula (Dahl and Nygaard, 1970), which evolved into the modern concept of inheritance, was defined in terms of textual concatenation of program blocks. However, this definition was informal, and only partially accounted for more sophisticated aspects of prefixing like the pseudo-variable this and virtual operations.

The most precise and widely used definition of inheritance is given by the operational semantics of object-oriented languages. The canonical operational semantics is the "method lookup" algorithm of Smalltalk:

> When a message is sent, the methods in the receiver's class are searched for one with a matching selector. If none is found, the methods in that class's superclass are searched next. The search continues up the superclass chain until a matching method is found. ...
>
> When a method contains a message whose receiver is self, the search for the method for that message begins in the instance's class, regardless of which class contains the method containing self. ...
>
> When a message is sent to super, the search for a method ... begins in the superclass of the class containing the method. The use of super allows a method to access methods defined in a superclass even if the methods have been overridden in the subclasses. (Goldberg and Robson, 1983, pp. 61–64)

Unfortunately, such operational definitions do not necessarily foster intuitive understanding. As a result, insight into the proper use and purpose of inheritance is often gained only through an "Aha!" experience (Borning and O'Shea, 1987).

Cardelli (1984) identifies inheritance with the subtype relation on record types: "a record type $\tau$ is a subtype (written $\leq$) of a record type $\tau'$ if $\tau$ has all the fields of $\tau'$, and possibly more, and the common fields of $\tau$ and $\tau'$ are in the $\leq$ relation." His work shows that a sound type-checking algorithm exists for strongly-typed, statically-scoped languages with inheritance, but it doesn't give their dynamic semantics.

More recently, McAllister and Zabih (1987) suggested a system of "boolean classes" similar to inheritance as used in knowledge representation. Stein (1987) focused on shared attributes and methods. Minsky and Rozenshtein (1987) characterized inheritance by "laws" regulating message sending. Although they express various aspects of inheritance, none of these presentations are convincing because they provide no verifiable evidence that the formal model corresponds to the form of inheritance actually used in object-oriented programming.

This paper presents a denotational model of inheritance. The model is based upon an intuitive explanation of the proper use and purpose of inheritance. It is well-known that inheritance is a mechanism for "differential programming" by allowing a new class to be defined by incremental modification of an existing class. We show that self-reference complicates the mechanism of incremental programming. In order for a derivation to have the same conceptual effect as direct modification, self-reference in the original definition must be changed to refer to the modified definition. This conceptual argument is useful for explaining the complex functionality of the pseudovariables self and super in Smalltalk.

Although the model was originally developed to describe inheritance in object-oriented languages, it shows that inheritance is a general mechanism that is applicable to any kind of recursive definition.

Essentially the same technical interpretation of inheritance was discovered independently by Reddy (1988). A closely related model was presented by Kamin (1988). However, Kamin describes inheritance as a global operation on programs, a formulation that blurs scope issues and inheritance.

These duplications, by themselves, are evidence for the validity of the model. This paper provides, in addition, a formal proof that the inheritance model is equivalent to the operational definition of inheritance quoted above.

Our denotational semantics of inheritance can be used as a basis for semantics-directed compiler generation for object-oriented langauges, as shown by Khoo and Sundaresh (1991).

In Section 2 we develop an intuitive motivation of inheritance. In Section 3 this intuition is formalized as a denotational model of inheritance. In Section 4 we demonstrate the correctness of the model by proving equivalence of two semantics of object-oriented systems, one based on the operational model and the other based upon the denotational model.

## 2 Motivating Inheritance

Inheritance is a mechanism for differential, or incremental, programming. Incremental programming is the construction of new program components by specifying how they differ from existing components. Incremental programming may be achieved by text editing, but this approach has a number of obvious disadvantages. A more disciplined approach to incremental programming is based upon using a form of "filter" to modify the external behavior of the original component. For example, to define a modified version of a function one simply defines a new function that performs some special computations and possibly calls the original function. This simple form of derivation is illustrated in Figure 1, where $P$ is the original function, $M$ is the modification, and the arrows represent invocation.

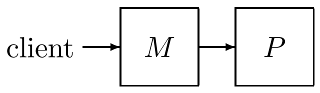

Figure 1: Derivation.

Incremental programming by explicit derivation is obviously more restrictive than text-editing; changes can be made either to the input passed to the original module or the output it returns, but the way in which the original works cannot be changed. Thus this form of derivation does not violate encapsulation (Snyder, 1986): the original structure can be replaced with an equivalent implementation and the derivation will have the same effect (text-editing is inherently unencapsulated).

However, there is one way in which this naive interpretation of derivation is radically different from text-editing: in the treatment of self-reference or recursion in the original structure. Figure 2 illustrates a naive derivation from a self-referential component.

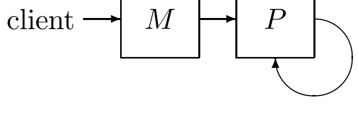

Figure 2: Naive derivation from a recursive structure.

Notice that the modification only affects external clients of the function — it does not modify the function's recursive calls. Thus naive derivation does not represent a true modification of the original component. To achieve the effect of a true modification of the original component, self-reference in the original function must be changed to refer to the modification, as illustrated in Figure 3.

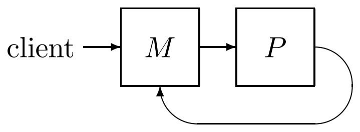

Figure 3: Inheritance.

This construction represents the essence of inheritance: it is a mechanism for deriving modified versions of recursive definitions.

## 3 A Model of Inheritance

This section develops the informal account of inheritance into a formal model of inheritance in object-oriented languages.

### 3.1 Self-referential Objects

Manipulation of self-reference is an essential feature of inheritance. Hence, the fixed point semantics of recursive definitions, developed by Scott (1976), provides the mathematical setting for the inheritance model. Introductions to fixed point semantics are given by Stoy (1977), Gordon (1979), and Schmidt (1986). The central theorem may be stated as follows.

**Theorem 1 (Fixed Point)** *If $D$ is a cpo and $f \in D \to D$ is continuous, then there is a least $x \in D$ such that $x = f(x)$. This $x$ is called the* least fixed point *of $f$, written $fix(f)$. It is given by $\bigsqcup_n f^n(\perp)$.*

Throughout, functions like $f$ will be called *generators*.

The following example illustrates a fixed point semantics of self-referential objects—essentially the standard interpretation of mutually recursive procedures. The example involves a simple class of points given in Figure 4. Points have x and y components to specify their location. The distFromOrig method computes their distance from the origin. closerToOrg is a method that takes another point object and returns *true* if the point is closer to the origin than the other point, and *false* otherwise.

```
class Point(a, b)
    method x = a
    method y = b
    method distFromOrig =
        sqrt(square(self.x) + square(self.y))
    method closerToOrg(p) =
        (self.distFromOrig < p.distFromOrig)
```

Figure 4: The class Point.

Objects are modeled as record values whose fields represent methods (Reddy, 1988, Cardelli and Wegner, 1985). The notation $\{l_1 \mapsto v_1, \ldots, l_n \mapsto v_n\}$ represents a record associating the value $v_i$ with label $l_i$. Records may in turn be viewed as finite functions from a domain of labels to a heterogeneous domain of values. Selection of the field $l$ from a record $m$ is achieved by applying the record to the label: $m.l$ or $m(l)$.

Class Point is modeled as a generator MakeGenPoint$(a, b)$, defined in Figure 5. MakeGenPoint takes the coordinates of the new point and returns a generator, whose fixed point is a point.

```
MakeGenPoint$(a, b)$ = $\lambda$ self .
    {   x $\mapsto a$,
        y $\mapsto b$,
        distFromOrig $\mapsto$
            sqrt(self.x$^2$ + self.y$^2$),
        closerToOrg $\mapsto$
            $\lambda p$ . (self.distFromOrig < $p$.distFromOrig)
        }
```

Figure 5: The generator associated with Point.

A point $(3, 4)$ is created as shown in Figure 6. The closerToOrg function takes a single argument which is assumed to be a point. Actually, all that is required is that it be a record with a distFromOrig component, whose value is a number.

$$
\begin{array}{rcl}
p & = & \mathrm{fix}(\mathsf{MakeGenPoint}(3,4)) \\
  & = & \{ \quad \mathsf{x} \mapsto 3, \\
  &   & \qquad \mathsf{y} \mapsto 4, \\
  &   & \qquad \mathsf{distFromOrig} \mapsto 5, \\
  &   & \qquad \mathsf{closerToOrg} \mapsto \\
  &   & \qquad\quad \lambda p\,.\,(5 < p.\mathsf{distFromOrig}) \\
  &   & \qquad \}
\end{array}
$$

Figure 6: A point at location (3,4).

### 3.2 Class Inheritance

Inheritance allows a new class to be defined by adding or replacing methods in an existing class. In the following example, the Point class is inherited to define a class of circles. Circles have a radius and thus a different notion of distance from the origin. The definition in Figure 7 gives only the differences between circles and points.

Inheritance is modeled as an operation on generators that yields a new generator. There are three aspects to this process: (1) the addition or replacement of methods, (2) the redirection of self-reference in the original generator to refer to the modified methods, and (3) the binding of super in the modification to refer to the original methods.

The modifications effected during class inheritance are naturally expressed as a record of methods to be combined with the inherited methods. The new methods $M$ and the original methods $O$ are combined into a new record $M \oplus O$ such that any method defined in $M$ replaces the corresponding method in $O$.

The modifications, however, are also defined in terms of the original methods (via super). In addition, the modifications refer to the resulting structure (via self). Thus a modification is naturally expressed as a function of two arguments, one representing self and the other representing super, that returns a record of locally defined methods. Such functions will be called *wrappers*. A wrapper contains just the information in the subclass definition. The wrapper for the subclass Circle is given in Figure 8.

```
class Circle(a, b, r) inherit Point(a, b)
    method radius = r

    method distFromOrig =
        max(super.distFromOrig − self.radius, 0)
```

Figure 7: The class Circle.

```
CircleWrapper = λ a, b, r . λ self . λ super . {
    {   radius ↦ r,
        distFromOrig ↦
           max(super.distFromOrig − self.radius, 0)
        }
```

Figure 8: The wrapper associated with Circle.

The appropriate operation on generators is *wrapper application*, which is defined as follows.

$$\boxed{\triangleright} : (\textbf{Wrapper} \times \textbf{Generator}) \rightarrow \textbf{Generator}$$
$$W \boxed{\triangleright} G = \lambda\,\mathsf{self}\,.\,(W(\mathsf{self})(G(\mathsf{self}))) \oplus G(\mathsf{self})$$

For an illustration of wrapper application, see Figure 9. A wrapper is applied to a generator to produce a new generator by first distributing self to both the wrapper and the original generator. Then the modifications defined by the wrapper are applied to the original record definition to produce a modification record. This is then combined with the original record using $\oplus$.

The generator associated with the class Circle can now be defined by wrapper application of CircleWrapper to MakeGenPoint, as shown in Figure 10. The figure also shows an expansion of the expression into a form that represents what one might write if circles had been defined without using inheritance. Note that distFromOrig has changed in such a way that

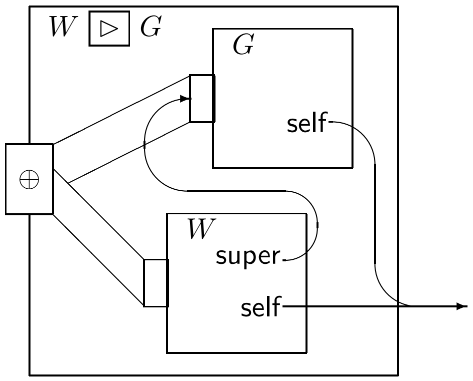

Figure 9: Wrapper application.

$$
\begin{aligned}
&\mathsf{MakeGenCircle} = \lambda\, a, b, r\,.\,\mathsf{CircleWrapper}(a, b, r)\ \boxed{\rhd}\ \mathsf{MakeGenPoint}(a, b)\\
&\quad = \lambda\, a, b, r\,.\,\lambda\,\mathsf{self}\,.\,(\mathsf{CircleWrapper}(a, b, r, \mathsf{self})(\mathsf{MakeGenPoint}(a, b, \mathsf{self})))\oplus\\
&\qquad\qquad\quad \mathsf{MakeGenPoint}(a, b, \mathsf{self})\\
&\quad = \lambda\, a, b, r\,.\,\lambda\,\mathsf{self}\,.\\
&\qquad \{\quad \mathsf{x} \mapsto a,\\
&\qquad\quad\ \ \mathsf{y} \mapsto b,\\
&\qquad\quad\ \ \mathsf{radius} \mapsto r,\\
&\qquad\quad\ \ \mathsf{distFromOrig} \mapsto\\
&\qquad\qquad\ \ \mathsf{max(sqrt(self.x^2 + self.y^2) - self.radius,\ 0)},\\
&\qquad\quad\ \ \mathsf{closerToOrg} \mapsto\\
&\qquad\qquad\ \ \lambda\, p\,.\,(\mathsf{self.distFromOrig} < p.\mathsf{distFromOrig})\\
&\qquad\quad\ \ \}
\end{aligned}
$$

Figure 10: The generator associated with Circle.

closerToOrg uses the notion of distance for circles, instead of the original one for points. Thus inheritance has achieved a consistent modification of the point class.

## 4 Correctness of the Model

To show the correctness of the inheritance model, we prove that it is equivalent to the definition of inheritance provided by the operational semantics of an object-oriented language. We introduce method systems as a useful framework in which to prove correctness. Two different semantics for method systems are then defined, based on the operational and denotational definitions of inheritance. Finally, we prove the equivalence of the two semantics.

### 4.1 Method Systems

Method systems are a simple formalization of object-oriented programming that support semantics based upon both the operational and the denotational models of inheritance. Method systems encompass only those aspects of object-oriented programming that are directly related to inheritance or method determination. As such, many important aspects are omitted, including instance variables, assignment, and object creation.

Method System Domains

| | | | |
|---|---|---|---|
| Instances | $\rho$ | $\in$ | **Instance** |
| Classes | $\kappa$ | $\in$ | **Class** |
| Messages | $m$ | $\in$ | **Key** |
| Primitives | $f$ | $\in$ | **Primitive** |
| Methods | $e$ | $\in$ | **Exp**::= $\mathsf{self} \mid \mathsf{super} \mid \mathsf{arg}$ |
| | | | $\mid e_1\ m\ e_2 \mid f(e_1,\ \ldots,\ e_q)$ |

Method System Operations

| | | |
|---|---|---|
| *class* | : | **Instance** $\rightarrow$ **Class** |
| *parent* | : | **Class** $\rightarrow$ (**Class** + ?) |
| *methods* | : | **Class** $\rightarrow$ **Key** $\rightarrow$ (**Exp** + ?) |

Figure 11: Syntactic domains and interconnections.

A method system may be understood as part of a snapshot of an object-oriented system. It consists of all the objects and relationships that exist at a given point during execution of an object-oriented program. The basic ontology for method systems includes instances, classes, and method descriptions, which are mappings from message keys to method expressions. Each object is an instance of a class. Classes have an associated method description and may inherit methods from other classes. These (flat) domains and their (monotone) interconnections are introduced in Figure 11. Figure 12 illustrates a method system.

The syntax of method expressions is defined by the **Exp** domain which defines a restricted language used to implement the behavior of objects. For simplicity, methods all have exactly one argument, referenced by the symbol **arg** within the body of the method. Self-reference is denoted by the symbol **self**, which may be returned as the value of a method, passed as an actual argument, or sent additional messages. A subclass method may invoke the previous definition of a redefined method with the expression **super**. Message-passing is represented by the expression $e_1\ m\ e_2$, in which the message consisting of the key $m$ and the argument $e_2$ is sent to the object $e_1$. Finally, primitive values and computations are represented by the expression $f(e_1,\ \ldots,\ e_q)$. If $q = 0$ then the primitive represents a constant.

*class* gives the class of an instance. Every instance has exactly one class, although a class may have many instances.

*parent* defines the inheritance hierarchy, which is required to be a tree. For any class $\kappa$, the value of *parent*$(\kappa)$ is the parent class of $\kappa$, or else $\bot_?$ if $\kappa$ is the root. **?** is a one-point domain consisting of only $\bot_?$. The use of (**Class** + ?) allows us to test monotonically whether a class is the root. Note that + denotes "separated" sum, so that the elements of (**Class** + ?) are (distinguished copies of) the elements of **Class**, the element $\bot_?$, and a new bottom element. We omit the injections into sum domains; the meaning of expressions, in particular $\bot_?$, is always unambiguously implied by the context.

*methods* specifies the local method expressions defined by a class. For any class $\kappa$ and any message key $m$, the value of *methods*$(\kappa)m$ is either an expression or $\bot_?$ if $\kappa$ doesn't define an expression for $m$. Let us assume that the root of the inheritance hierarchy doesn't define any methods. Note that inheritance allows instances of a class to respond to more than the locally defined methods.

In the following two sections we give the method system both a conventional method lookup semantics and a denotational semantics. To do so we need a notion of a *program*, and we choose the simplest possible one. Informally, it is "create an instance and send a message to it". Both the semantics give a denotation of such an instance; the denotation define the

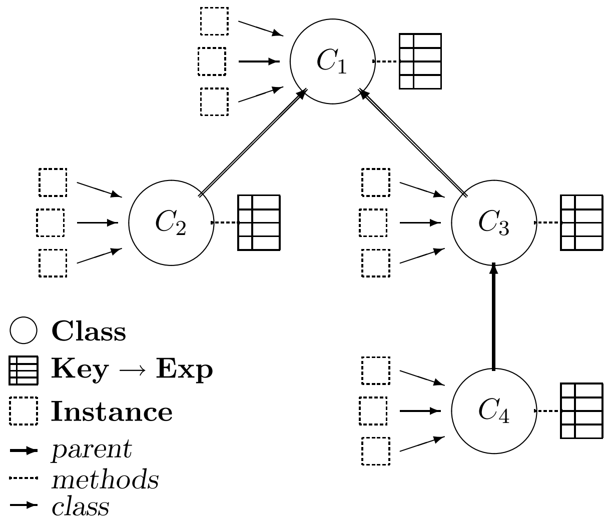

Figure 12: A method system.

result of sending a message to the instance.

## 4.2 Method Lookup Semantics

The method lookup semantics given in Figure 14 closely resembles the implementation of method lookup in object-oriented languages like Smalltalk (Goldberg and Robson, 1983). It is given in a denotational style due to the abstract nature of method systems. A more traditional operational semantics is not needed because of the absence of updatable storage.

The domains used to represent the behavior an of instance are defined in Figure 13. A behavior is a mapping from message keys to *functions* or $\perp_?$. This is clearly contrasted with the methods of a class, which are given by a mapping from message keys to *expressions* or $\perp_?$. Thus a behavior is a semantic entity, while methods are syntactic. Another difference between the behavior of an instance and its class's methods is that the behavior contains a function for every message the class handles, while methods associate an expression only with messages that are different from the class's parent. In the rest of this paper, $\perp$ (without subscript) denotes the bottom element of **Behavior**.

Semantic Domains

$$\begin{array}{rcll} & & \mathbf{Number} & \\ \alpha & \in & \mathbf{Value} & = \mathbf{Behavior} + \mathbf{Number} \\ \sigma, \pi & \in & \mathbf{Behavior} & = \mathbf{Key} \rightarrow (\mathbf{Fun} + ?) \\ \phi & \in & \mathbf{Fun} & = \mathbf{Value} \rightarrow \mathbf{Value} \end{array}$$

$$\boxed{\begin{array}{l} root : \mathbf{Class} \rightarrow \mathbf{Boolean} \\ root(\kappa) = [\lambda\, \kappa' \in \mathbf{Class}\, .\, false, \\ \qquad\qquad \lambda\, v \in ?\, .\, true \\ \qquad\qquad ](parent(\kappa)) \end{array}}$$

Figure 13: Semantic domains and *root*.

The semantics also uses an auxiliary function *root*, defined in Figure 13, that determines whether a class is the root of the inheritance hierarchy. **Boolean** is the flat three-point domain of truth values. $[f, g]$ denotes the case analysis of two functions $f \in D_f \rightarrow D$ and $g \in D_g \rightarrow D$ with result in the domain $D$, mapping $x \in D_f + D_g$ to $f(x)$ if $x \in D_f$ or to $g(x)$ if $x \in D_g$.

Sending a message $m$ to an instance $\rho$ is performed by looking up the message in the instance's class. The lookup process yields a function that takes a message key and an actual argument and computes the value of the message send.

Performing message $m$ in a class $\kappa$ on behalf of an instance $\rho$ involves searching the sequence of class parents until a method is found to handle the message. This method is then evaluated. In *lookup*, the instance and message remain constant, while the class argument is recursively bound to each of the parents in sequence. At each stage there are two possibilities: (1) the message key has an associated method expression in class $\kappa$, in which case it is evaluated, and (2) the method is not defined, in which case a recursive call is made to *lookup* after computing the parent of the class. Note that when *do* is called from *lookup*, then the value of the argument $\kappa$ is the class in which the method was found. This class need not be the class of the instance $\rho$, hence $\rho$ is a separate argument to *lookup* so that it later can be used in the semantics of **self**. The tail-recursion in *lookup* would be replaced by iteration in a real interpreter.

$$
\begin{array}{l}
\mathit{send} : \mathbf{Instance} \to \mathbf{Behavior} \\
\mathit{send}(\rho) = \mathit{lookup}(\mathit{class}(\rho))\rho \\
\\
\mathit{lookup} : \mathbf{Class} \to \mathbf{Instance} \to \mathbf{Behavior} \\
\mathit{lookup}(\kappa)\rho = \lambda\, m \in \mathbf{Key}\,. \\
\qquad [\lambda\, e \in \mathbf{Exp}\,.\, \mathit{do}[\![\, e \,]\!]\rho\kappa, \\
\qquad\ \lambda\, v \in \,?\,.\, \mathbf{if}\ \mathit{root}(\kappa) \\
\qquad\qquad\quad \mathbf{then}\ \bot_? \\
\qquad\qquad\quad \mathbf{else}\ \mathit{lookup}(\mathit{parent}(\kappa))\rho m \\
\qquad ](\mathit{methods}(\kappa)m) \\
\\
\mathit{do} : \mathbf{Exp} \to \mathbf{Instance} \to \mathbf{Class} \to \mathbf{Fun} \\
\mathit{do}[\![\, \textsf{self} \,]\!]\rho\kappa = \lambda\, \alpha \in \mathbf{Value}\,.\, \mathit{send}(\rho) \\
\mathit{do}[\![\, \textsf{super} \,]\!]\rho\kappa = \lambda\, \alpha \in \mathbf{Value}\,.\, \mathit{lookup}(\mathit{parent}(\kappa))\rho \\
\mathit{do}[\![\, \textsf{arg} \,]\!]\rho\kappa = \lambda\, \alpha \in \mathbf{Value}\,.\, \alpha \\
\mathit{do}[\![\, e_1\ m\ e_2 \,]\!]\rho\kappa = \lambda\, \alpha \in \mathbf{Value}\,.\, (\mathit{do}[\![\, e_1 \,]\!]\rho\kappa\alpha)m(\mathit{do}[\![\, e_2 \,]\!]\rho\kappa\alpha) \\
\mathit{do}[\![\, f(e_1,\ \ldots,\ e_q) \,]\!]\rho\kappa = \\
\qquad \lambda\, \alpha \in \mathbf{Value}\,.\, (\mathit{id} f)(\mathit{do}[\![\, e_1 \,]\!]\rho\kappa\alpha,\ \ldots,\ \mathit{do}[\![\, e_q \,]\!]\rho\kappa\alpha)
\end{array}
$$

$$\begin{tikzcd}
\mathit{send} \arrow[r] & \mathit{lookup} \arrow[r] \arrow[loop above] & \mathit{do} \arrow[loop above] \arrow[ll, bend left] \arrow[l, bend left]
\end{tikzcd}$$

Figure 14: The method lookup semantics.

Evaluation of methods is complicated by the need to interpret occurrences of **self** and **super**. The *do* function has three extra arguments, besides the expression being evaluated: the original instance $\rho$ that received the message whose method is being evaluated, the class $\kappa$ in which the method was found, and an actual argument $\alpha$. The expression **self** evaluates to the behavior of the original instance. The expression **super** requires a continuation of the method search starting from the superclass of the class in which the method occurs. The expression **arg** evaluates to $\alpha$. The expression $e_1\ m\ e_2$ evaluates to the result of applying the behavior of the object denoted by $e_1$ to $m$ and the meaning of the argument $e_2$. Finally, the semantics of primitive values and computations is given using the operation *id*, which maps primitives to constants or functions. Since we omitted a detailed description of **Primitive**, we also omit that of *id*.

For an illustration of the semantics of **super**, consider three classes A, B, and C, where B inherits A, and C inherits B. Assume that a method $m$ is defined in A, overridden in B, and simply inherited in C. Suppose that the message $m$ is sent to an instance of A. Method lookup for $m$ will yield the method in B. Should this method send the message $m$ to **super**, however, the method yielded will the one in A, not that in the superclass of C.

One important aspect of the method lookup semantics is that the functions are mutually recursive, because *do* contains calls to *send* and *lookup*.

## 4.3 Denotational Semantics

The denotational semantics based on generator modification given in Figure 16 uses two additional domains representing behavior generators and wrappers, defined in Figure 15. A formal definition of $\oplus$ is also given in Figure 15.

The behavior of an instance is defined as the fixed point of the generator associated with its class. The generator specifies a self-referential behavior, and its fixed point is that behavior. The generator of the root class produces a behavior in which all messages are undefined.

Generator Semantics Domains

$$\begin{array}{rcl}
\textbf{Generator} & = & \textbf{Behavior} \to \textbf{Behavior} \\
\textbf{Wrapper} & = & \textbf{Behavior} \to \textbf{Behavior} \to \textbf{Behavior}
\end{array}$$

$$\boxed{\begin{array}{l}
\oplus : (\textbf{Behavior} \times \textbf{Behavior}) \to \textbf{Behavior} \\
r_1 \oplus r_2 = \lambda\, m \in \textbf{Key} \,.\, [\lambda\, \phi \in \textbf{Fun} \,.\, \phi, \\
\qquad\qquad\qquad\qquad\quad \lambda\, v \in ?\,.\, r_2(m) \\
\qquad\qquad\qquad\qquad\quad ]r_1(m)
\end{array}}$$

Figure 15: Semantic domains and $\oplus$.

The generator of a class that isn't the root is created by modifying the generator of the class's parent. The modifications to be made are found in the wrapper of the class, which is a semantic entity derived from the block of

$$\begin{array}{l}
\mathit{behave} : \mathbf{Instance} \rightarrow \mathbf{Behavior} \\
\mathit{behave}(\rho) = \mathrm{fix}(\mathit{gen}(\mathit{class}(\rho))) \\
\\
\mathit{gen} : \mathbf{Class} \rightarrow \mathbf{Generator} \\
\mathit{gen}(\kappa) = \mathbf{if}\ \mathit{root}(\kappa) \\
\qquad\qquad \mathbf{then}\ \lambda\,\sigma \in \mathbf{Behavior}\,.\,\lambda\,m \in \mathbf{Key}\,.\,\bot_{?} \\
\qquad\qquad \mathbf{else}\ \mathit{wrap}(\kappa)\ \boxed{\triangleright}\ \mathit{gen}(\mathit{parent}(\kappa)) \\
\\
\mathit{wrap} : \mathbf{Class} \rightarrow \mathbf{Wrapper} \\
\mathit{wrap}(\kappa) = \lambda\,\sigma \in \mathbf{Behavior}\,.\,\lambda\,\pi \in \mathbf{Behavior}\,.\,\lambda\,m \in \mathbf{Key}\,. \\
\qquad [\lambda\,e \in \mathbf{Exp}\,.\,\mathit{eval}[\![\,e\,]\!]\sigma\pi \\
\qquad\ \ \lambda\,v \in ?\,.\,\bot_{?} \\
\qquad ]\mathit{methods}(\kappa)m \\
\\
\mathit{eval} : \mathbf{Exp} \rightarrow \mathbf{Behavior} \rightarrow \mathbf{Behavior} \rightarrow \mathbf{Fun} \\
\mathit{eval}[\![\,\mathsf{self}\,]\!]\sigma\pi = \lambda\,\alpha \in \mathbf{Value}\,.\,\sigma \\
\mathit{eval}[\![\,\mathsf{super}\,]\!]\sigma\pi = \lambda\,\alpha \in \mathbf{Value}\,.\,\pi \\
\mathit{eval}[\![\,\mathsf{arg}\,]\!]\sigma\pi = \lambda\,\alpha \in \mathbf{Value}\,.\,\alpha \\
\mathit{eval}[\![\,e_1\ m\ e_2\,]\!]\sigma\pi = \\
\qquad \lambda\,\alpha \in \mathbf{Value}\,.\,(\mathit{eval}[\![\,e_1\,]\!]\sigma\pi\alpha)m(\mathit{eval}[\![\,e_2\,]\!]\sigma\pi\alpha) \\
\mathit{eval}[\![\,f(e_1,\ \ldots,\ e_q)\,]\!]\sigma\pi = \\
\qquad \lambda\,\alpha \in \mathbf{Value}\,.\,(\mathit{id}\,f)(\mathit{eval}[\![\,e_1\,]\!]\sigma\pi\alpha,\ \ldots,\ \mathit{eval}[\![\,e_q\,]\!]\sigma\pi\alpha)
\end{array}$$

$$\begin{tikzcd}
\mathit{behave} \arrow[r] & \mathit{gen} \arrow[r] \arrow[loop above] & \mathit{wrap} \arrow[r] & \mathit{eval} \arrow[loop above]
\end{tikzcd}$$

Figure 16: The denotational semantics.

syntactic method expressions defined by the class. These modifications are effected by the inheritance operator $\boxed{\triangleright}$. Recall that $\boxed{\triangleright}$: (1) distributes a fresh self to both the wrapper and the parent generator, (2) applies the wrapper to the parent generator to produce a modification behavior, and (3) combines the modification and the parent behavior, using $\oplus$.

The function *wrap* computes the wrapper of a class as a mapping from messages to the evaluation of the corresponding method, or to $\bot_{?}$. A wrapper has two behavioral arguments, one used for self-reference, and the other for reference to the parent behavior (i.e. the behavior being ‘wrapped’). These arguments may be understood as representing the behavior of **self** and the behavior of **super**. In the definitions, the behavior for **self** is named $\sigma$ and the one for **super** is named $\pi$.

A method is always evaluated in the context of a behavior for **self** (represented by $\sigma$) and **super** (represented by $\pi$). The evaluation of the corresponding expressions, **self** and **super**, is therefore simple. The evaluation of the other expressions is essentially the same as in the method lookup semantics.

Note that each of the functions in the denotational semantics is recursive only within itself: there is no mutual recursion among the functions, except that which is achieved by the explicit fixed point.

For illustration of the denotational semantics, let us briefly reexamine the examples in section 3. The meaning of class Point(a,b) in figure 4 is essentially *gen*(Point(a,b)), called MakeGenPoint(a,b) in figure 5. The meaning of an instance of Point(a,b) is shown in figure 6. The wrapper for the class Circle(a,b,r) in figure 7 is essentially *wrap*(Circle(a,b,r)), called CircleWrapper in figure 8. Finally, the generator associated with Circle(a,b,r) is essentially *gen*(Circle(a,b,r)), called MakeGenCircle in figure 10.

## 4.4 Equivalence

The method lookup semantics and the denotational semantics are equivalent because they assign the same behavior to an instance. This proposition is captured by theorem 2.

**Theorem 2** $\mathit{send} = \mathit{behave}$

In the proof of the theorem we use an “intermediate semantics” defined in Figure 17 and inspired by the one used by Mosses and Plotkin (1987) in their proof of limiting completeness. The semantics uses $n \in \mathbf{Nat}$, the flat domain of natural numbers.

The intermediate semantics resembles the method lookup semantics but differs in that each of the syntactic domains of instances, classes, and expressions has a whole family of semantic equations, indexed by natural numbers. The intuition behind the definition is that $\mathit{send}'_n\rho$ allows $(n-1)$ evaluations of **self** before it stops and gives $\bot$. $\mathit{send}'_n\rho$ is defined in terms of $\mathit{send}'_{n-1}\rho$ via $\mathit{lookup}'_n$ and $\mathit{do}'_n$ because the **self** expression evaluates to the result of $\mathit{send}'_{n-1}\rho$, which allows one less evaluation of **self**. (The values of $\mathit{lookup}'_0\kappa\rho$ and $\mathit{do}'_0[\![\, e \,]\!]\rho\kappa$ are irrelevant; let them be $\bot$ and $\lambda\,\alpha\,.\,\bot$.)

```
$send' : \mathbf{Nat} \to \mathbf{Instance} \to \mathbf{Behavior}$
$send'_0(\rho) = \bot$
$\mathit{if}\ n > 0\ \mathit{then}$
  $send'_n(\rho) = lookup'_n(class(\rho))\rho$

$lookup' : \mathbf{Nat} \to \mathbf{Class} \to \mathbf{Instance} \to \mathbf{Behavior}$
$lookup'_0 \kappa\rho = \bot$
$\mathit{if}\ n > 0\ \mathit{then}$
  $lookup'_n \kappa\rho = \lambda\, m \in \mathbf{Key}\,.$
        $[\lambda\, e \in \mathbf{Exp}\,.\, do'_n[\![\, e \,]\!]\rho\kappa,$
         $\lambda\, v \in ?\,.\,\mathbf{if}\ root(\kappa)$
                    $\mathbf{then}\ \bot_?$
                    $\mathbf{else}\ lookup'_n(parent(\kappa))\rho m$
        $](methods(\kappa)m)$

$do' : \mathbf{Nat} \to \mathbf{Exp} \to \mathbf{Instance} \to \mathbf{Class} \to \mathbf{Fun}$
$do'_0[\![\, e \,]\!]\rho\kappa = \lambda\, \alpha \in \mathbf{Value}\,.\,\bot$
$\mathit{if}\ n > 0\ \mathit{then}$
  $do'_n[\![\, \textsf{self} \,]\!]\rho\kappa = \lambda\, \alpha \in \mathbf{Value}\,.\, send'_{n-1}\rho$
  $do'_n[\![\, \textsf{super} \,]\!]\rho\kappa = \lambda\, \alpha \in \mathbf{Value}\,.\, lookup'_n(parent(\kappa))\rho$
  $do'_n[\![\, \textsf{arg} \,]\!]\rho\kappa = \lambda\, \alpha \in \mathbf{Value}\,.\, \alpha$
  $do'_n[\![\, e_1\ m\ e_2 \,]\!]\rho\kappa =$
          $\lambda\, \alpha \in \mathbf{Value}\,.\, (do'_n[\![\, e_1 \,]\!]\rho\kappa\alpha)m(do'_n[\![\, e_2 \,]\!]\rho\kappa\alpha)$
  $do'_n[\![\, f(e_1,\ \ldots,\ e_q) \,]\!]\rho\kappa =$
          $\lambda\, \alpha \in \mathbf{Value}\,.\, (id f)(do'_n[\![\, e_1 \,]\!]\rho\kappa\alpha,\ \ldots,\ do'_n[\![\, e_q \,]\!]\rho\kappa\alpha)$
```

Figure 17: The intermediate semantics.

The following four lemmas state useful properties of the intermediate semantics. Here we only outline their proofs, leaving the full proofs to appendix A.

**Lemma 1** *If* $n > 0$ *then*

$$do'_n[\![\, e \,]\!]\rho\kappa = eval[\![\, e \,]\!](send'_{n-1}\rho)(lookup'_n(parent(\kappa))\rho)$$

PROOF: By induction on the structure of $e$, using the definitions of $do'$ and *eval*.

**Lemma 2** *If $n > 0$ then*

$$\mathit{lookup}'_n \kappa \rho = \mathit{gen}(\kappa)(\mathit{send}'_{n-1}\rho)$$

PROOF: By induction on the number of ancestors of $\kappa$, using the definitions of *gen*, $\boxed{\triangleright}$, $\oplus$, and *wrap*, Lemma 1, and the definition of *lookup*$'$.

**Lemma 3** $\mathit{send}'_n \rho = (\mathit{gen}(\mathit{class}(\rho)))^n(\bot)$

PROOF: By induction on $n$, using Lemma 2 and the definition of *send*$'$.

**Lemma 4** *send$'$, lookup$'$, and do$'$ are monotone functions of the natural numbers with the usual ordering.*

PROOF: Immediate from Lemma 1–3.

Lemma 4 expresses that the family of $\mathit{send}'_n$'s is an increasing sequence of functions.

**Definition 1**

$$\begin{array}{l} \mathit{interpret} : \textbf{Instance} \to \textbf{Behavior} \\ \mathit{interpret} = \bigsqcup_n (\mathit{send}'_n) \end{array}$$

The following three propositions express the relations among the method lookup semantics, the intermediate semantics, and the denotational semantics.

**Proposition 1** *interpret = behave*

PROOF: The following proof uses the definition of *interpret*, Lemma 3, the fixed point theorem, and the definition of *behave*. The fixed point theorem is applicable since $\mathit{gen}(\mathit{class}(\rho))$ is continuous (we omit the proof).

$$\begin{array}{rcl} \mathit{interpret}(\rho) & = & \bigsqcup_n (\mathit{send}'_n(\rho)) \\ & = & \bigsqcup_n (\mathit{gen}(\mathit{class}(\rho)))^n(\bot) \\ & = & \mathrm{fix}(\mathit{gen}(\mathit{class}(\rho))) \\ & = & \mathit{behave}(\rho) \end{array}$$

QED

**Proposition 2** *send* $\sqsupseteq$ *behave*

PROOF: The following facts have proofs analogous to those of Lemma 1–2 (we omit the proofs).

1. $do[\![ e ]\!]\rho\kappa = eval[\![ e ]\!](send(\rho))(lookup(parent(\kappa))\rho)$
2. $lookup(\kappa)\rho = gen(\kappa)(send(\rho))$

From the definition of *send* and the second fact we get $send(\rho) = lookup(class(\rho))\rho = gen(class(\rho))(send(\rho))$. Hence $send(\rho)$ is a fixed point of $gen(class(\rho))$. The definition of *behave* expresses that $behave(\rho)$ is the least fixed point of $gen(class(\rho))$; thus $send(\rho) \sqsupseteq behave(\rho)$. QED

**Proposition 3** $send \sqsubseteq interpret$

PROOF: The functions defined in the method lookup semantics are mutually recursive. Their meaning is the least fixed point of the generator $g$ defined in the obvious way, as outlined below.

$$\begin{array}{rcl} D & = & (\textbf{Instance} \rightarrow \textbf{Behavior}) \\ & & \times(\textbf{Class} \rightarrow \textbf{Instance} \rightarrow \textbf{Behavior}) \\ & & \times(\textbf{Exp} \rightarrow \textbf{Instance} \rightarrow \textbf{Class} \rightarrow \textbf{Fun}) \end{array}$$

Let $g : D \rightarrow D$ be defined by

$$g(s, l, d) = (\lambda\, i \in \textbf{Instance}\,.\, l(class(\rho))\rho, \ldots, \ldots)$$

The three components of $g$ correspond to *send*, *lookup*, and *do*, and they are defined similarly, except that they refer to each other instead of *send*, *lookup*, and *do*. Now we can prove by induction on $n$ that

$$g^n(\bot_D) \sqsubseteq (send'_n, lookup'_n, do'_n)$$

In the base case, where $n = 0$, the inequality holds trivially. Then assume that the inequality holds for $(n-1)$, where $n > 0$. The following proof of the induction step uses the monotonicity of $g$ (we omit the proof), the induction hypothesis, and Lemma 4.

$$\begin{array}{rcl} g^n(\bot_D) & = & g(g^{n-1}(\bot_D)) \\ & \sqsubseteq & g(send'_{n-1}, lookup'_{n-1}, do'_{n-1}) \\ & \sqsubseteq & (send'_n, lookup'_n, do'_n) \end{array}$$

The following calculation uses the fixed point theorem, which is applicable since $g$ is continuous (we omit the proof).

$$\begin{array}{rcl} (\mathit{send}, \mathit{lookup}, \mathit{do}) & = & \mathrm{fix}(g) \\ & = & \bigsqcup_n g^n(\bot_D) \\ & \sqsubseteq & \bigsqcup_n(\mathit{send}'_n, \mathit{lookup}'_n, \mathit{do}'_n) \end{array}$$

In particular, we have $\mathit{send} \sqsubseteq \bigsqcup_n(\mathit{send}'_n) = \mathit{interpret}$.
QED

PROOF of Theorem 2: Combine Propositions 1–3. QED

## 5 Conclusion

A denotational semantics of inheritance was presented, using a general notation that is applicable to the analysis of different object-oriented languages (Cook, 1989). The semantics was supported by an intuitive explanation of inheritance as a mechanism for incremental programming that derives modified versions of recursive structures. An explanation of the binding of self- and super-reference was given at this conceptual level. To provide evidence for the correctness of the model, it was proven equivalent to the most widely accepted definition of inheritance, the operational method lookup semantics used in object-oriented languages.

In comparing the denotational semantics with the operational semantics, the denotational one does not seem to be much simpler. It may even be argued that it is a great deal more complex, because it requires an understanding of fixed points. The primary advantage of the denotational semantics is the intuitive explanation it provides. It suggests that inheritance may be useful for other kinds of recursive structures, like types and functions, in addition to classes. It also reveals that inheritance, while a natural extension of existing mechanisms, does provide expressive power not found in conventional languages by allowing more flexible use of the fixed point function.

*Acknowledgement.* The authors would like to thank Peter Wegner for the original motivation for an equivalence proof, Peter Mosses for helpful comments on the second part of the paper, and John Mitchell for comments on an earlier draft.

## References

Borning, A. H., and O'Shea, T. (1987), Deltatalk: An empirically and aesthetically motivated simplification of the Smalltalk-80 language, *in* "European Conference on Object-Oriented Programming", pp. 1–10.

Cardelli, L. (1984), A semantics of multiple inheritance, *in* "Semantics of Data Types", LNCS 173, Springer-Verlag, pp. 51–68.

Cardelli, L., and Wegner, P. (1985), On understanding types, data abstraction, and polymorphism, *Computing Surveys*, 17(4), pp. 471–522.

Cook, W. (1989), "A Denotational Semantics of Inheritance", PhD thesis, Brown University.

Dahl, O.-J., and Nygaard, K. (1970), "The SIMULA 67 Common Base Language".

Goldberg, A., and Robson, D. (1983), "Smalltalk-80: the Language and Its Implementation", Addison-Wesley.

Gordon, M. J. C. (1979), "The Denotational Description of Programming Languages", Springer-Verlag.

Kamin, S. (1988), Inheritance in Smalltalk-80: A denotational definition, *in* "Proceedings, 15th Symposium on Principles of Programming Languages", pp. 80–87.

Khoo, S. C., and Sundaresh, R. S. (1991), Compiling inheritance using partial evaluation, *in* "Proceedings, ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation", pp. 211–222.

McAllester, D., and Zabih, R. (1987), Boolean Classes, *in* "Proceedings, ACM Conference on Object-Oriented Programming: Systems, Languages and Applications", pp. 417–423.

Minsky, N., and Rozenshtein, D. (1987), A law-based approach to object-oriented programming, *in* "Proceedings, ACM Conference on Object-Oriented Programming: Systems, Languages and Applications", pp. 482–493.

Mosses, P. D., and Plotkin, G. D. (1987), On proving limiting completeness, *SIAM Journal of Computing*, 16, pp. 179–194.

Reddy, U. S. (1988), Objects as closures: Abstract semantics of object-oriented languages, *in* "Proceedings, ACM Conference on Lisp and Functional Programming", pp. 289–297.

Schmidt, D. A. (1986), "Denotational Semantics: A Methodology for Language Development", Allyn & Bacon.

Scott, D. A. (1976), Data types as lattices. *SIAM Journal*, 5(3), pp. 522–586.

Snyder, A. (1986), Encapsulation and inheritance in object-oriented programming languages, *in* "Proceedings, ACM Conference on Object-Oriented Programming: Systems, Languages and Applications", pp. 38–45.

Stein, L. A. (1987), Delegation is inheritance, *in* "Proceedings, ACM Conference on Object-Oriented Programming: Systems, Languages and Applications", pp. 138–146.

Stoy, J. (1977), "Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics", MIT Press.

## Appendix A: Proofs of Lemmas

In this appendix we give the full proofs of Lemmas 1–4. A summary of definitions, except the three semantics, is given in appendix B.

PROOF of Lemma 1: Recall that we want to prove, for $n > 0$, that

$$do'_n[\![\, e \,]\!]\rho\kappa = eval[\![\, e \,]\!](send'_{n-1}\rho)(lookup'_n(parent(\kappa))\rho)$$

by induction on the structure of $e$. The base case is proved as follows.

$$\begin{aligned}
do'_n[\![\, \textsf{self} \,]\!]\rho\kappa\alpha &= send'_{n-1}\rho \\
&= eval[\![\, \textsf{self} \,]\!](send'_{n-1}\rho)(lookup'_n(parent(\kappa))\rho)\alpha \\
do'_n[\![\, \textsf{super} \,]\!]\rho\kappa\alpha &= lookup'_n(parent(\kappa))\rho \\
&= eval[\![\, \textsf{super} \,]\!](send'_{n-1}\rho)(lookup'_n(parent(\kappa))\rho)\alpha \\
do'_n[\![\, \textsf{arg} \,]\!]\rho\kappa\alpha &= \alpha \\
&= eval[\![\, \textsf{arg} \,]\!](send'_{n-1}\rho)(lookup'_n(parent(\kappa))\rho)\alpha
\end{aligned}$$

The induction step is proven below using the abbreviation $\pi = lookup'_n(parent(\kappa))\rho$.

$$\begin{aligned}
do'_n[\![\, e_1 \ m \ e_2 \,]\!]\rho\kappa\alpha &= do'_n[\![\, e_1 \,]\!]\rho\kappa\alpha m(do'_n[\![\, e_2 \,]\!]\rho\kappa\alpha) \\
&= eval[\![\, e_1 \,]\!](send'_{n-1}\rho)\pi\alpha m(eval[\![\, e_2 \,]\!](send'_{n-1}\rho)\pi\alpha) \\
&= eval[\![\, e_1 \ m \ e_2 \,]\!](send'_{n-1}\rho)\pi\alpha \\
do'_n[\![\, f(e_1, \ \ldots, \ e_q) \,]\!]\rho\kappa\alpha &= (id f)(do'_n[\![\, e_1 \,]\!]\rho\kappa\alpha, \ \ldots, \ do'_n[\![\, e_q \,]\!]\rho\kappa\alpha) \\
&= (id f)(eval[\![\, e_1 \,]\!](send'_{n-1}\rho)\pi\alpha, \ \ldots, \\
&\qquad eval[\![\, e_q \,]\!](send'_{n-1}\rho)\pi\alpha) \\
&= eval[\![\, f(e_1, \ \ldots, \ e_q) \,]\!](send'_{n-1}\rho)\pi\alpha
\end{aligned}$$

QED

PROOF of Lemma 2: Recall that we want to prove $lookup'_n\kappa\rho = gen(\kappa)(send'_{n-1}\rho)$, where $n > 0$, by induction on the number of ancestors of $\kappa$. In the base case, where $\kappa$ is the root, both sides evaluate to $(\lambda\, m \in \mathbf{Key} \,.\, \bot_?)$ because $\kappa$ doesn't define any methods. Then assume that the lemma holds for $parent(\kappa)$. The proof of the induction step given below uses the definition of $gen$ ($root(\kappa)$ is false), the definition of $\boxed{\triangleright}$, the induction hypothesis, the definitions of $\oplus$ and *wrap*, the properties of case analysis, Lemma 1, and the definition of *lookup*$'$ ($root(\kappa)$ is false). Note that we use the abbreviation $\pi = lookup'_n(parent(\kappa))\rho$.

$$\begin{aligned}
gen(\kappa)(send'_{n-1}\rho) &= (wrap(\kappa) \boxed{\rhd} gen(parent(\kappa)))(send'_{n-1}\rho) \\
&= (wrap(\kappa)(send'_{n-1}\rho)(gen(parent(\kappa))(send'_{n-1}\rho))) \\
&\qquad \oplus (gen(parent(\kappa))(send'_{n-1}\rho)) \\
&= (wrap(\kappa)(send'_{n-1}\rho)\pi) \oplus \pi \\
&= \lambda\, m \in \mathbf{Key}\,.\,[\lambda\, \phi \in \mathbf{Fun}\,.\,\phi, \\
&\qquad\qquad\qquad \lambda\, v \in ?\,.\,\pi m \\
&\qquad\qquad\qquad ](wrap(\kappa)(send'_{n-1}\rho)\pi m) \\
&= \lambda\, m \in \mathbf{Key}\,.\,[\lambda\, \phi \in \mathbf{Fun}\,.\,\phi, \\
&\qquad\qquad\qquad \lambda\, v \in ?\,.\,\pi m \\
&\qquad\qquad\qquad ]([\lambda\, e \in \mathbf{Exp}\,.\,eval[\![ e ]\!](send'_{n-1}\rho)\pi, \\
&\qquad\qquad\qquad\quad \lambda\, v \in ?\,.\,\bot_? \\
&\qquad\qquad\qquad\quad ](methods(\kappa)m)) \\
&= \lambda\, m \in \mathbf{Key}\,.\,[\lambda\, e \in \mathbf{Exp}\,.\,eval[\![ e ]\!](send'_{n-1}\rho)\pi, \\
&\qquad\qquad\qquad \lambda\, v \in ?\,.\,\pi m \\
&\qquad\qquad\qquad ](methods(\kappa)m) \\
&= \lambda\, m \in \mathbf{Key}\,.\,[\lambda\, e \in \mathbf{Exp}\,.\,do'_n[\![ e ]\!]\rho\kappa, \\
&\qquad\qquad\qquad \lambda\, v \in ?\,.\,\pi m \\
&\qquad\qquad\qquad ](methods(\kappa)m) \\
&= lookup'_n(\kappa)\rho
\end{aligned}$$

QED

PROOF of Lemma 3: Recall that we want to prove $send'_n\rho = (gen(class(\rho)))^n(\bot)$ by induction on $n$. In the base case, where $n = 0$, both sides evaluate to $\bot$. Then assume that the lemma holds for $(n-1)$, where $n > 0$. The following proof of the induction step uses the associativity of function composition, the induction hypothesis, Lemma 2, and the definition of *send*$'$.

$$\begin{aligned}
(gen(class(\rho)))^n(\bot) &= gen(class(\rho))((gen(class(\rho)))^{n-1}(\bot)) \\
&= gen(class(\rho))(send'_{n-1}\rho) \\
&= lookup'_n(class(\rho))\rho \\
&= send'_n\rho
\end{aligned}$$

QED

PROOF of Lemma 4: We must prove that $\mathit{send}'$, $\mathit{lookup}'$, and $\mathit{do}'$ are monotone functions of the natural numbers with the usual ordering. From Lemma 3 it follows that $\mathit{send}'$ is monotone. Then, if $n \leq m$ we have $\mathit{lookup}'_n \kappa\rho = \mathit{gen}(\kappa)(\mathit{send}'_{n-1}\rho) \sqsubseteq \mathit{gen}(\kappa)(\mathit{send}'_{m-1}\rho) = \mathit{lookup}'_m \kappa\rho$ using Lemma 2, that $\mathit{send}'$ is monotone, and Lemma 2 again. Finally, we can in the same way prove that $\mathit{do}'$ is monotone using Lemma 1, that $\mathit{send}'$ and $\mathit{lookup}'$ are monotone, and Lemma 1 again. Note that we also use that $\mathit{gen}(\kappa)$ and $\mathit{eval}[\![\, e \,]\!]$ are monotone (we omit the proof).
QED

## Appendix B: Summary of Definitions

$$\boxed{\triangleright} : (\textbf{Wrapper} \times \textbf{Generator}) \rightarrow \textbf{Generator}$$
$$W \boxed{\triangleright} G = \lambda\,\mathsf{self}\,.\,(W(\mathsf{self})(G(\mathsf{self}))) \oplus G(\mathsf{self})$$

Method System Domains

| | | | |
|---|---|---|---|
| Instances | $\rho$ | $\in$ | **Instance** |
| Classes | $\kappa$ | $\in$ | **Class** |
| Messages | $m$ | $\in$ | **Key** |
| Primitives | $f$ | $\in$ | **Primitive** |
| Methods | $e$ | $\in$ | **Exp**$::= \textsf{\textbf{self}} \mid \textsf{\textbf{super}} \mid \textsf{\textbf{arg}}$ |
| | | | $\mid e_1\ m\ e_2 \mid f(e_1,\ \ldots,\ e_q)$ |

Method System Operations

| | | |
|---|---|---|
| *class* | : | $\textbf{Instance} \rightarrow \textbf{Class}$ |
| *parent* | : | $\textbf{Class} \rightarrow (\textbf{Class} + ?)$ |
| *methods* | : | $\textbf{Class} \rightarrow \textbf{Key} \rightarrow (\textbf{Exp} + ?)$ |

Semantic Domains

| | | | |
|---|---|---|---|
| | | **Number** | |
| $\alpha$ | $\in$ | **Value** | $= \textbf{Behavior} + \textbf{Number}$ |
| $\sigma, \pi$ | $\in$ | **Behavior** | $= \textbf{Key} \rightarrow (\textbf{Fun} + ?)$ |
| $\phi$ | $\in$ | **Fun** | $= \textbf{Value} \rightarrow \textbf{Value}$ |

$$\begin{aligned} &root : \textbf{Class} \rightarrow \textbf{Boolean} \\ &root(\kappa) = [\lambda\,\kappa' \in \textbf{Class}\,.\,\mathit{false}, \\ &\qquad\quad \lambda\,v \in ?\,.\,\mathit{true} \\ &\qquad\ ](parent(\kappa)) \end{aligned}$$

Generator Semantics Domains

| | | |
|---|---|---|
| **Generator** | = | $\textbf{Behavior} \rightarrow \textbf{Behavior}$ |
| **Wrapper** | = | $\textbf{Behavior} \rightarrow \textbf{Behavior} \rightarrow \textbf{Behavior}$ |

$$\begin{aligned} &\oplus : (\textbf{Behavior} \times \textbf{Behavior}) \rightarrow \textbf{Behavior} \\ &r_1 \oplus r_2 = \lambda\,m \in \textbf{Key}\,.\,[\lambda\,\phi \in \textbf{Fun}\,.\,\phi, \\ &\qquad\qquad\qquad\qquad\quad \lambda\,v \in ?\,.\,r_2(m) \\ &\qquad\qquad\qquad\qquad\ ]r_1(m) \end{aligned}$$