ISSN 0105-8517

A Denotational Semantics of Inheritance and its Correctness

William Cook
Jens Palsberg

DAIMI PB – 284
August 1989

AARHUS UNIVERSITY
COMPUTER SCIENCE DEPARTMENT
Ny Munkegade 116 – DK 8000 Aarhus C – DENMARK
Telephone: + 45 86 12 71 88 *Telex: 64767 aausci dk*

Abstract

This paper presents a denotational model of inheritance. The model is based on an intuitive motivation of the purpose of inheritance. 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. Although it was originally developed to explain inheritance in object-oriented languages, the model shows that inheritance is a general mechanism that may be applied to any form of recursive definition.

Current addresses:

William Cook, Hewlett-Packard Laboratories, P.O. Box 10490 Palo Alto, CA 94303-0969. cook@hplabs.hp.com.

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

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 [5], 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. [6, 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 [1].

Cardelli [2] identifies inheritance with the subtype relation on record types: "a record type τ is a subtype (written $\leq$) of a record type τ' if τ has all the fields of τ', and possibly more, and the common fields of τ and τ' 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 [9] suggested a system of "boolean classes" similar to inheritance as used in knowledge representation. Stein [16] focused on shared attributes and methods. Minsky and Rozenshtein [10] 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 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 [12]. A closely related model was presented by Kamin [8]. 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.

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.

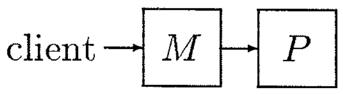

Figure 1: Derivation.

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 [15]: 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.

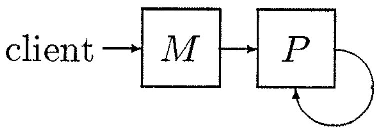

Figure 2: Naive derivation from a recursive structure.

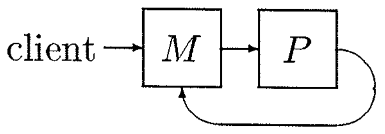

Figure 3: Inheritance.

Notice that the modification only affects external clients of the function — it does not modify the function's recursive calls. This 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 class must be changed to refer to the modification, as illustrated in figure 3.

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

3 A Model of Inheritance

This section develops the informal account of inheritance into a formal model of inheritance in object-oriented languages. Since manipulation of self-reference was identified as a central feature of inheritance, inheritance is explained within a traditional semantics of self-reference or recursion.

3.1 Fixed-Point Semantics

The fixed-point semantics of recursive programs, developed by Scott [14], provides the mathematical setting for the inheritance model. The technique of fixed point semantics is merely illustrated here; thorough introductory explanations are given by Stoy [17], Gordon [7], Scott [14], and Schmidt [13].

To illustrate the use of fixed points for the analysis of recursive programs, consider the following definition of the factorial function:

$$\mathit{fact} = \lambda n\,.\,\textbf{if}\ n = 1\ \textbf{then}\ 1\ \textbf{else}\ n \times \mathit{fact}(n-1)$$

From a mathematical point of view, this definition is not very satisfactory because *fact* appears on both sides of the equation: the definition is merely an equation that *fact* must satisfy. There is no guarantee that any function satisfying this equation exists, and even less that there is a unique one.

This problem is solved by fixed-point analysis, which indicates how to construct the denotation of such self-referential definitions by "solving" the equation given above. To use fixed-point techniques, the recursive definition is transformed into a non-recursive form. First, the body of the function is converted into an explicit abstraction, or function, in which the parameter f is substituted for *fact*:

$\mathit{FACT} = \lambda f\,.\,\lambda n\,.\,\textbf{if}\ n = 1\ \textbf{then}\ 1\ \textbf{else}\ n \times f(n-1)$

Functions derived in this way will be called ***generators***. FACT is a ***functional***, or mapping from functions to functions. Its definition is not recursive, because 'FACT' does not appear in its body. The formal parameter f represents the function to call in order to compute the factorial of numbers less than n, if needed. The original definition of *fact* is then given in terms of FACT:

$\mathit{fact} = \mathit{FACT}(\mathit{fact})$

But now *fact* is defined as a value that is unchanged when FACT is applied. Such a value is called a ***fixed point***, of FACT. Under certain conditions, it is possible to define a particular ***fixed point***, the ***least fixed point***, of any function by using the ***fixed-point function***, fix. The fixed-point function has the property that if $f = \mathrm{fix}(F)$, then $F(f) = f$. Another way to express the least fixed point, which is utilized later in the paper, is as the limit of the series $\bot, F(\bot), F(F(\bot)), \ldots$. For fix to work the function F must be ***continuous*** (our domains are cpos), a condition that is satisfied by our definitions in the rest of the paper.

3.2 Self-referential Objects

The following example illustrates how fixed-point semantics is used to describe the behavior of objects with mutually recursive methods. This is essentially the standard interpretation of mutually recursive procedures [18, 19]. 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.

Objects are modeled as record values whose fields represent methods [12, 3]. The notation $\{ l_1 \mapsto v_1, \ldots, l_n \mapsto v_n \}$ represents a record

```
class Point(a, b)
    method x = a
    method y = b
    method distFromOrig = sqrt(self.x^2 + self.y^2)
    method closerToOrg(p) = (self.distFromOrig < p.distFromOrig)
```

Figure 4: The class Point.

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 $m(l)$.

By using this model, the class Point is represented as a function that creates objects. References to self in Point are explained using fixed points. A function MakeGenPoint is defined to construct points and supply them with appropriate methods. Since points are self-referential, they are explained as the fixed point of the "generator" of the methods. MakeGenPoint, defined in Figure 5, is a function that takes the coordinates of the new point and returns a generator, whose fixed point is a point.

$$
\begin{aligned}
&\mathsf{MakeGenPoint}(a, b) = \lambda\,\mathsf{self}\,.\\
&\quad \{\ \mathsf{x} \mapsto a,\\
&\qquad \mathsf{y} \mapsto b,\\
&\qquad \mathsf{distFromOrig} \mapsto \mathsf{sqrt(self.x^2 + self.y^2)},\\
&\qquad \mathsf{closerToOrg} \mapsto \lambda p\,.\,(\mathsf{self.distFromOrig} < p.\mathsf{distFromOrig})\,\}
\end{aligned}
$$

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.

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

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

3.3 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.

This form of inheritance is modeled as an operation on generators. 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.

The other aspect of inheritance that must be formalized is the change of self-reference in inherited methods. The methods to be inherited are contained in a generator, a function whose argument is used for self-reference in the methods. The result of inheritance should be a new class

```
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.

$$\begin{aligned}&\mathsf{CircleWrapper} = \lambda\, a, b, r\, .\, \lambda\, \mathsf{self}\, .\, \lambda\, \mathsf{super}\, .\\ &\{\ \ \mathsf{radius} \mapsto r,\\ &\quad\ \mathsf{distFromOrig} \mapsto \mathsf{max(super.distFromOrig - self.radius,\ 0)}\,\}\end{aligned}$$

Figure 8: The wrapper associated with Circle.

definition, that is, a new generator.

This mechanism is provided by ***wrapper application.*** 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 mechanism of wrapper application is defined by the infixed written inheritance operator $\boxed{\triangleright}$ as follows:

$$W\ \boxed{\triangleright}\ P = \lambda\, \mathsf{self}\, .\, (W(\mathsf{self})(P(\mathsf{self}))) \oplus P(\mathsf{self})$$

Note that two occurrences of P refer to the same behavioral denotation, and do not indicate that the parent is instantiated twice. The mechanism of wrapper application is illustrated in Figure 9.

The generator which represents the class Circle can now be defined by wrapper application of CircleWrapper to MakeGenPoint, as shown in Figure 10. The figure also shows the partial 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 closerToOrg will use the notion of distance for circles, instead of the original one for points. Thus inheritance has achieved a consistent modification of the point class.

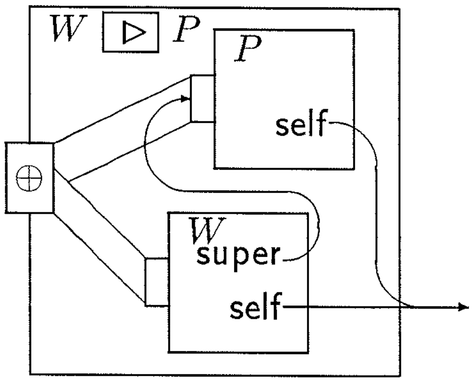

Figure 9: Wrapper application.

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

Figure 10: The generator associated with Circle.

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 which support semantics based upon both the operational and the

Method System Domains

Instances	ρ	$\in$	**Instance**
Classes	κ	$\in$	**Class**
Messages	m	$\in$	**Key**
Primitives	f	$\in$	**Primitive**
Methods	e	$\in$	**Exp**:= **self** | **super** | **arg**
			| $e_1\ m\ e_2$ | $f(e_1,\ \ldots,\ e_q)$

Method System Operations

class	:	$\mathbf{Instance} \to \mathbf{Class}$
parent	:	$\mathbf{Class} \to (\mathbf{Class} + ?)$
methods	:	$\mathbf{Class} \to \mathbf{Key} \to (\mathbf{Exp} + ?)$

Figure 11: Syntactic domains and interconnections.

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.

A method system may be understood as part of a snap-shot of an object-oriented system. It consists of all the objects and relationships which 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 interconnections are defined in Figure 11. An illustration of a method system is provided in Figure 12.

The syntax of method expressions is defined by the **Exp** domain. This domain defines a restricted language used to implement the behavior of objects. For simplicity, methods all have exactly one argument, which is 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

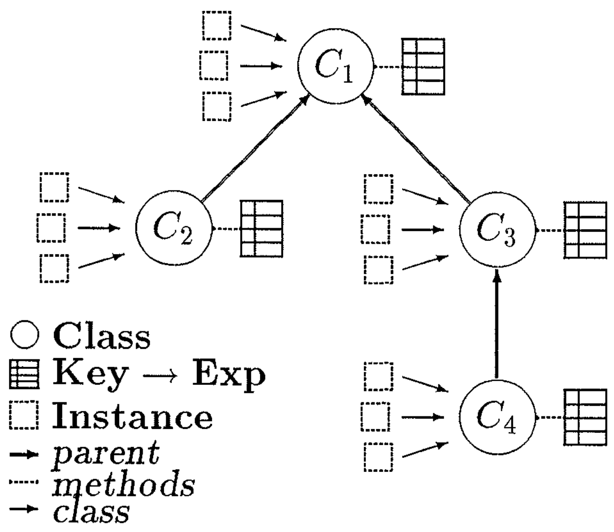

Figure 12: A method system.

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 κ, the value of *parent*(κ) is the parent class of κ, or else $\perp_?$ if κ is the root. **?** is a 1-point domain, consisting of only $\perp_?$. 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 $\perp_?$, and a new bottom element. We will omit the injections into sum domains; the meaning of expressions, in particular $\perp_?$, will always be unambiguously implied by the context.

methods specifies the local method expressions defined by a class. For any class κ and any message key m, the value of *methods*$(\kappa)m$ is either an expression or $\perp_?$ if κ 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. Both define the result of sending a message to an 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 [6]. 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 $\bot_?$. This is clearly contrasted with the methods of a class, which are given by a mapping from message keys to ***expressions*** or $\bot_?$. Thus a behavior is a semantic entity, while methods are syntactic. Another difference between the behavior of an instance and its class' 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' parent. In the rest of this paper, $\bot$ (without subscript) denotes the bottom element of **Behavior**.

Semantic Domains

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

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

Figure 13: Semantic domains and *root*.

The semantics also uses an auxiliary function *root* which determines whether a class is the root of the inheritance hierarchy, defined in Figure 13. **Boolean** is the flat 3-point domain of truth values. $[f, g]$ denotes the case analysis of two functions $f \in D_f \to t$ and $g \in D_g \to t$ with result

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

Figure 14: The method lookup semantics.

in type t, 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 ρ is performed by looking up the message in the instance's class. The lookup process results in 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 κ on behalf of an instance ρ 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 κ, 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. The tail-recursion in *lookup* would be replaced by iteration in a real interpreter.

Evaluation of methods is complicated by the necessity to interpret occurrences of **self** and **super**. The *do* function has three extra arguments, besides the expression being evaluated: the original instance ρ which received the message whose method is being evaluated, the class κ in which the method was found, and an actual argument α. 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 α. 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.

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

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

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

Figure 15: Semantic Domains and $\oplus$.

The generator of a class which isn't the root is created by modifying the generator of the class' parent. The modifications to be made are found in the wrapper of the class, which is a semantic entity derived from the block of syntactic method expressions defined by the class. These modifications are effected by the inheritance operator $\boxed{\triangleright}$.

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

Figure 16: The denotational semantics.

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 σ and the one for **super** is named π.

The evaluation of a method is always performed in the context of a behavior for **self** (represented by σ) and **super** (represented by π). 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 are recursive only within themselves: there is no mutual recursion among the functions, except that which is achieved by the explicit fixed-point.

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 1.

Theorem 1 *send = behave*

In the proof of the theorem we use an "intermediate semantics", defined in Figure 17, that is inspired by the one used by Mosses and Plotkin [11] 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 $(n-1)$ occurrences of **self** on its way before it stops and gives $\bot$. In fact, we define $\mathit{send}'_n\rho$ in terms of $\mathit{send}'_{n-1}\rho$ via lookup'_n and do'_n because the **self** expression evaluates to the result of $\mathit{send}'_{n-1}\rho$ which allows one less occurrence of **self**. (Noticing that the values of $\mathit{lookup}'_0\kappa\rho$ and $\mathit{do}'_0[\![e]\!]\rho\kappa\alpha$ are irrelevant, we let them be $\bot$.)

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

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

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

PROOF: By induction in 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 in the number of ancestors of κ, 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 in n, using Lemma 2 and the definition of send'.

$$
\begin{array}{l}
\mathit{send}' : \mathbf{Nat} \rightarrow \mathbf{Instance} \rightarrow \mathbf{Behavior} \\
\mathit{send}'_0(\rho) = \bot \\
\mathit{send}'_n(\rho) = \mathit{lookup}'_n(\mathit{class}(\rho))\rho \qquad n > 0 \\
\\
\mathit{lookup}' : \mathbf{Nat} \rightarrow \mathbf{Class} \rightarrow \mathbf{Instance} \rightarrow \mathbf{Behavior} \\
\mathit{lookup}'_0\kappa\rho = \bot \\
\mathit{if}\ n > 0\ \mathit{then} \\
\mathit{lookup}'_n\kappa\rho = \lambda\, m \in \mathbf{Key}\,.\,[\lambda\, e \in \mathbf{Exp}\,.\,\mathit{do}'_n[\![\, e \,]\!]\rho\kappa, \\
\qquad\qquad\qquad\qquad\qquad\quad \lambda\, v \in ?\,.\,\mathbf{if}\ \mathit{root}(\kappa) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{then}\ \bot_? \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{else}\ \mathit{lookup}'_n(\mathit{parent}(\kappa))\rho m \\
\qquad\qquad\qquad\qquad\qquad\quad](\mathit{methods}(\kappa)m) \\
\\
\mathit{do}' : \mathbf{Nat} \rightarrow \mathbf{Exp} \rightarrow \mathbf{Instance} \rightarrow \mathbf{Class} \rightarrow \mathbf{Fun} \\
\mathit{do}'_0[\![\, e \,]\!]\rho\kappa = \lambda\, \alpha\,.\,\bot \\
\mathit{if}\ n > 0\ \mathit{then} \\
\quad \mathit{do}'_n[\![\, \mathsf{self} \,]\!]\rho\kappa = \lambda\, \alpha\,.\,\mathit{send}'_{n-1}\rho \\
\quad \mathit{do}'_n[\![\, \mathsf{super} \,]\!]\rho\kappa = \lambda\, \alpha\,.\,\mathit{lookup}'_n(\mathit{parent}(\kappa))\rho \\
\quad \mathit{do}'_n[\![\, \mathsf{arg} \,]\!]\rho\kappa = \lambda\, \alpha\,.\,\alpha \\
\quad \mathit{do}'_n[\![\, e_1\ m\ e_2 \,]\!]\rho\kappa = \lambda\, \alpha\,.\,(\mathit{do}'_n[\![\, e_1 \,]\!]\rho\kappa\alpha)m(\mathit{do}'_n[\![\, e_2 \,]\!]\rho\kappa\alpha) \\
\quad \mathit{do}'_n[\![\, f(e_1,\ \ldots,\ e_q) \,]\!]\rho\kappa = \lambda\, \alpha\,.\,f(\mathit{do}'_n[\![\, e_1 \,]\!]\rho\kappa\alpha,\ \ldots,\ \mathit{do}'_n[\![\, e_q \,]\!]\rho\kappa\alpha)
\end{array}
$$

Figure 17: The intermediate semantics.

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 send'_n's is an increasing sequence of functions.

Definition 1
$\mathit{interpret} : \mathbf{Instance} \rightarrow \mathbf{Behavior};\ \mathit{interpret} = \sqcup_n(\mathit{send}'_n)$

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***.

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

QED

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

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

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

From the definition of ***send*** and the second fact we get $\mathit{send}(\rho) = \mathit{lookup}(\mathit{class}(\rho))\rho = \mathit{gen}(\mathit{class}(\rho))(\mathit{send}(\rho))$. Hence $\mathit{send}(\rho)$ is a fixed-point of $\mathit{gen}(\mathit{class}(\rho))$. The definition of ***behave*** expresses that $\mathit{behave}(\rho)$ is the least fixed-point of $\mathit{gen}(\mathit{class}(\rho))$, thus $\mathit{send}(\rho) \sqsupseteq \mathit{behave}(\rho)$. 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 which is defined in the obvious way as outlined below.

$$\begin{aligned} D = \; & (\mathbf{Instance} \to \mathbf{Behavior}) \\ & \times(\mathbf{Class} \to \mathbf{Instance} \to \mathbf{Behavior}) \\ & \times(\mathbf{Exp} \to \mathbf{Instance} \to \mathbf{Class} \to \mathbf{Fun}) \end{aligned}$$

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

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

Now we can prove by induction in n that $g^n(\bot_D) \sqsubseteq (\mathit{send}'_n, \mathit{lookup}'_n, \mathit{do}'_n)$. In the base case, where $n = 0$, the inequality trivially holds. Then assume that the inequality holds for $(n-1)$, where $n > 0$. The following proof of the induction step uses the associativity of function composition, the induction hypothesis, and Lemma 4.

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

Now $(\mathit{send}, \mathit{lookup}, \mathit{do}) = \mathrm{fix}(g) = \sqcup_n g^n(\bot_D) \sqsubseteq \sqcup_n(\mathit{send}'_n, \mathit{lookup}'_n, \mathit{do}'_n)$ and in particular $\mathit{send} \sqsubseteq \sqcup_n(\mathit{send}'_n) = \mathit{interpret}$. QED

PROOF of theorem 1: 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 [4]. The semantics was supported by an intuitive explanation of inheritance as a mechanism for incremental programming that simulate destructive modification. 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. Another important advantage of the denotational semantics is a demonstration 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 Mosses for helpful comments on the second part of the paper, and John Mitchell for comments on an earlier draft.

References

[1] Alan H. Borning and Tim O'Shea. Deltatalk: An empirically and aesthetically motivated simplification of the Smalltalk-80 language. In *European Conf. on Object-Oriented Programming*, pages 1–10, 1987.

[2] Luca Cardelli. A semantics of multiple inheritance. In *Semantics of Data Types, LNCS 173*, pages 51–68. Springer-Verlag, 1984.

[3] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[4] William Cook. *A Denotational Semantics of Inheritance.* PhD thesis, Brown University, 1989.

[5] Ole-Johan Dahl and Kristen Nygaard. *The SIMULA 67 Common Base Language.* 1970.

[6] Adele Goldberg and Dave Robson. *Smalltalk-80: the Language and Its Implementation.* Addison-Wesley, 1983.

[7] Michael J. C. Gordon. *The Denotational Description of Programming Languages.* Springer-Verlag, 1979.

[8] Samuel Kamin. Inheritance in Smalltalk-80: A denotational definition. In *Proc. of Conf. on Principles of Programming Languages*, pages 80–87, 1988.

[9] D. McAllester and R. Zabih. Boolean classes. In *Proc. ACM Conf. on Object-Oriented Programming: Systems, Languages and Applications*, pages 417–423, 1987.

[10] N. Minsky and D. Rozenshtein. A law-based approach to object-oriented programming. In *Proc. ACM Conf. on Object-Oriented Programming: Systems, Languages and Applications*, pages 482–493, 1987.

[11] Peter Mosses and Gordon Plotkin. On proving limiting completeness. *SIAM Journal of Computing*, 16:179–194, 1987.

[12] Uday S. Reddy. Objects as closures: Abstract semantics of object-oriented languages. In *Proc. ACM conf. on Lisp and Functional Programming*, pages 289–297, 1988.

[13] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn & Bacon, 1986.

[14] Dana S. Scott. Data types as lattices. *SIAM Journal*, 5(3):522–586, 1976.

[15] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proc. ACM Conf. on Object-Oriented Programming: Systems, Languages and Applications*, pages 38–45, 1986.

[16] Lynn Andrea Stein. Delegation is inheritance. In *Proc. ACM Conf. on Object-Oriented Programming: Systems, Languages and Applications*, pages 138–146, 1987.

[17] Joseph Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics.* MIT Press, 1977.

[18] Robert D. Tennent. On a new approach to representation-independent data classes. *Acta Informatica*, 8:315–324, 1977.

[19] Robert D. Tennent. Denotational semantics of Hoare classes. Unpublished manuscript, 1982.

Appendix: Proofs of Lemmas

In this appendix we give the full proofs of Lemma 1–4.

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 in the structure of e. The base case is proved as follows.

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

$$\begin{aligned}
&do'_n[\![\, \textsf{super} \,]\!]\rho\kappa\alpha \\
&\quad = lookup'_n(parent(\kappa))\rho
\end{aligned}$$

$$= \mathit{eval}[\![\,\textbf{super}\,]\!](\mathit{send}'_{n-1}\rho)(\mathit{lookup}'_n(\mathit{parent}(\kappa))\rho)\alpha$$

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

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

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

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

QED

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

$$\begin{aligned}
&\mathit{gen}(\kappa)(\mathit{send}'_{n-1}\rho)\\
&\quad= (\mathit{wrap}(\kappa)\ \boxed{\triangleright}\ \mathit{gen}(\mathit{parent}(\kappa)))(\mathit{send}'_{n-1}\rho)\\
&\quad= (\mathit{wrap}(\kappa)(\mathit{send}'_{n-1}\rho)(\mathit{gen}(\mathit{parent}(\kappa))(\mathit{send}'_{n-1}\rho)))\\
&\qquad\quad \oplus(\mathit{gen}(\mathit{parent}(\kappa))(\mathit{send}'_{n-1}\rho))\\
&\quad= (\mathit{wrap}(\kappa)(\mathit{send}'_{n-1}\rho)\pi) \oplus \pi
\end{aligned}$$

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

QED

PROOF of Lemma 3: Recall that we want to prove $\mathit{send}'_n\rho = (\mathit{gen}(\mathit{class}(\rho)))^n(\bot)$ by induction in 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*′.

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

QED

PROOF of Lemma 4: We must prove that *send*′, *lookup*′, and *do*′ are monotone functions of the natural numbers with the usual ordering. From Lemma 3 it follows that *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 *send*′ is monotone, and Lemma 2 again. Finally, we can in the same way prove that *do*′ is monotone using Lemma 1, that *send*′ and *lookup*′ are monotone, and Lemma 1 again.
QED