Objects as Closures: Abstract Semantics of Object Oriented Languages[1]

Uday S. Reddy[2]
Department of Computer Science University of Illinois at Urbana-Champaign
Net: reddy@a.cs.uiuc.edu

Abstract

We discuss denotational semantics of object–oriented languages, using the concept of *closure* widely used in (semi) functional programming to encapsulate side effects. It is shown that this denotational framework is adequate to explain classes, instantiation, and inheritance in the style of Simula as well as SMALLTALK-80. This framework is then compared with that of Kamin, in his recent denotational definition of SMALLTALK-80, and the implications of the differences between the two approaches are discussed.

1 Introduction

Object-oriented languages, such as SMALLTALK–80[3] [9], have recently received a lot of attention. However, the term "object-oriented" does not seem to have a widely accepted meaning. It is sometimes used to refer to the presence of *data objects with local state*, sometimes to the notion of *class inheritance*, and sometimes to the specific notion of inheritance in Smalltalk which involves a kind of "dynamic binding". The first of these notions, viz., objects with local state, has long been used in the functional programming community to encapsulate "side effects" whenever they were necessary [1, 11]. These are sometimes losely referred to as *closures*. A closure is essentially a function or a data structure containing functions with some local bindings to values or storage locations. In describing the semantics of object oriented languages, it seems natural that such a notion of closure should play a central role.

SMALLTALK and other object oriented languages, of course, go much beyond data objects with local states. They allow classes to be defined, objects to be created as instances of classes, class descriptions to refer to the receiving object in terms of *self*, and subclasses to be derived from superclasses. Whether all these concepts can be explained in terms of closures is an interesting question. If so, the denotational semantics of object oriented languages can be defined in terms of closures. This paper answers this question in the affirmative and presents such a denotational semantics.

In a recent paper [10], Kamin presented a denotational semantics for SMALLTALK–80 using a different framework. Here objects are interpreted denotationally as pairs of local environments and references to class denotations. The denotations of classes are defined independently of the objects that receive messages. The essential objection we raise against this scheme is that the semantics is not sufficiently abstract. From the point of view of the user of an object, an object simply responds to a set of messages. So, the meaning of an object should simply be

[1] This paper is to appear in 1988 *ACM Conference on Lisp and Functional Programming*, Snowbird, July, 1988.

[2] This research is supported in part by NSF grant CCR-87-00988 and NASA grant NAG-1-613.

[3] "SMALLTALK–80" is a trademark of ParcPlace Systems. We use here a language called "SmallTalk" (with different capitalization) as an abstraction of SMALLTALK–80.

an environment binding message names to their methods (*message environments*). There is no need for the local environment of an object to appear in its denotation. Our presentation of the semantics interprets objects precisely as message environments.

To present the semantics, we discuss a series of small abstract languages. Firstly, *ObjectTalk* is a language in which objects can be defined, but no classes. In the second language, *ClassTalk*, classes can be defined and objects can be created as instances of classes. The third language, called *InheritTalk*, provides subclasses to be defined by inheriting from other classes. The bindings of messages used by superclasses are not affected by inheritance. The inheritance of Simula [7] and C++ [18] work in this fashion. Finally, we define a language called *SmallTalk*, which implements inheritance in the style of SMALLTALK–80, by rebinding messages in subclasses. It is shown that ClassTalk and InheritTalk are extensions of ObjectTalk, i.e., they do not alter the denotations used in earlier languages. But, SmallTalk requires a radical restructuring of the denotations.

In addition, we restate the semantics of Kamin in our notation and formally establish their correspondence by showing a homomorphism from Kamin's semantic domains to ours. There does not exist a homomorphism in the other direction, because Kamin's domains contain strictly more information (and hence are less abstract) than ours.

2 Denotational Framework

Our style of presentation will be to consider a series of little abstract languages with increasingly more expressive power. For obvious reasons, we will not treat a full language, but only those portions which are of interest to object-oriented programming. To set the context, let us first give some examples of syntactic constructs:

$$
\begin{array}{rcl}
x, y & \in & \textit{variable} \\
e & \in & \textit{expression}
\end{array}
$$

$$
\begin{array}{rcl}
e & ::= & x \\
e & ::= & \mathsf{valof}\ e \\
e & ::= & x := e \\
e & ::= & \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2
\end{array}
$$

Here, we have only two kinds of syntactic objects *variable* and *expression*, and three kinds of expression constructs. For pedagogical reasons, we use the dereferencing operator valof to access the contents of a location. (It allows us to use a single semantic function, rather than two separate ones for the *l-* and *r-values* of expressions).

Conventionally, the meaning of an expression [17] is of the type

$$\textit{env} \rightarrow \textit{state} \rightarrow \textit{val} \times \textit{state}.$$

So, an expression valuation $[\![e]\!]\eta\sigma$ is some $\langle v, \sigma' \rangle$. The bindings of free variables in e, which may be values or locations, are obtained from η, and the contents of locations are obtained from the state σ. Our semantic domains and a sampler of semantic definitions are given below:

$$
\begin{array}{rcll}
x, y & \in & \mathit{variable} & \\
\alpha & \in & \mathit{loc} & \\
v, w & \in & \mathit{val} & = \mathit{basicval} + \mathit{loc} + \cdots \\
\eta & \in & \mathit{env} & = \mathit{variable} \to \mathit{val} \\
\sigma & \in & \mathit{state} & = \mathit{loc} \to \mathit{val}
\end{array}
$$

$$
[\![-]\!] \;:\; \mathit{env} \to \mathit{state} \to (\mathit{val} \times \mathit{state})
$$

$$
\begin{array}{rcl}
[\![x]\!]\eta\sigma & = & \langle \eta x, \sigma \rangle \\
[\![\mathsf{valof}\ e]\!]\eta\sigma & = & \textbf{let } \langle \alpha, \sigma' \rangle = [\![e]\!]\eta\sigma \\
 & & \textbf{in } \alpha \in \mathit{loc} \to \langle \sigma'\alpha, \sigma' \rangle;\ ? \\
[\![x := e]\!]\eta\sigma & = & \textbf{let } \alpha = \eta\, x \\
 & & \qquad \langle v, \sigma_1 \rangle = [\![e]\!]\eta\sigma \\
 & & \textbf{in } \alpha \in \mathit{loc} \to \langle v, \sigma_1[\alpha \to v] \rangle;\ ? \\
[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!]\eta\sigma & = & \textbf{let } \langle v_1, \sigma_1 \rangle = [\![e_1]\!]\eta\sigma \\
 & & \textbf{in } [\![e_2]\!]\,(\eta[x \to v_1])\,\sigma
\end{array}
$$

Let us make a few comments about our notation. The symbol ? denotes an error value. We do not elaborate its meaning any further. (See [17] for a detailed discussion). Environments and states are finite functions, and we often need to update them (like in the semantics of assignment above). The notation

$$f[x \to v]$$

means a copy of the function f that maps x to v, leaving everything else unchanged. We also use the notation

$$f[x_1 \to v_1, \ldots, x_k \to v_k]$$

when we need to update the mapping of several values simultaneously. A third notational device is

$$f; f'$$

which means updating of f with all the bindings of f' (note: f' should be a finite mapping). The symbol $\eta_\perp$ denotes the empty environment and $\sigma_\perp$ deontes the empty state.

The notion of *closure* arises from the fact that expressions may have free (nonlocal) variables. The type $\mathit{env} \to \mathit{state} \to (\mathit{val} \times \mathit{state})$ shows that an expression valuation depends on an environment and a state. Given both, the value of the expression is fixed. Now, consider a procedure valued expression with free variables, e.g.,

$$\mathsf{let}\ f() = (x := \mathsf{valof}\ y)\ \mathsf{in}\ f$$

The "value" (i.e. the *val* part in the above type) of such an expression is, in turn, of the type

$$\mathit{procedure} = \mathit{state} \to \mathit{val}^* \to (\mathit{val} \times \mathit{state})$$

It can be applied in some state σ to some tuple of values $\overline{v}$, producing a result value and a new state. The environment at the point of its application does not affect its meaning. Thus, if the definition of f is evaluated (not applied) in an environment η with $\eta\, x = \alpha_1$ and $\eta\, y = \alpha_2$, then the value of f is

$$c \equiv \lambda\sigma.\ \lambda\langle\rangle.\ \langle(\sigma\ \alpha_2),\ \sigma[\alpha_1 \rightarrow (\sigma\ \alpha_2)]\rangle$$

The variables x and y have been replaced by their bindings α_1 and α_2, and this procedure will forever transfer the contents of the location α_2 to the location α_1. A procedure value, such as c, is called a *closure*. The expression of which it is a value may have had free variables. But, they have all been eliminated before we obtain the value. The closure itself now does not "depend" on any variables. We can also conceive of languages (like Lisp) in which the meaning of the procedure depends on the environment at the point of application. Then, a procedure value should take as its parameter. Such a procedure value is not a closure.

Another programming language feature concerned with closures is the declaration of mutable variables in local contexts. To make this precise, let us add another construct to our example language:

$$e ::= \mathsf{local}\ x;\ e\ \mathsf{end}$$

Its semantics is given by

$$\begin{array}{l}
[\![\mathsf{local}\ x;\ e\ \mathsf{end}]\!]\eta\sigma = \\
\quad \textbf{let}\ \ \alpha = \mathit{newloc}\ \sigma \\
\qquad\qquad \text{– new location for } x \\
\qquad\ \ \sigma_1 = \mathit{extend}\ \sigma\ \alpha \\
\qquad\qquad \text{– allocation of the location} \\
\qquad\ \ \eta_1 = \eta[x \rightarrow \alpha] \\
\qquad\qquad \text{– local environment for } e \\
\quad \textbf{in}\ [\![e]\!]\ \eta_1\ \sigma_1
\end{array} \tag{1}$$

This sequence of definitions arises so often in this paper that we introduce a new function *alloc* for it:

$$\begin{array}{lll}
\mathit{alloc}\ \sigma\ x = & \textbf{let} & \alpha = \mathit{newloc}\ \sigma \\
& & \sigma_1 = \mathit{extend}\ \sigma\ \alpha \\
& & \eta_0 = \eta_\perp[x \rightarrow \alpha] \\
& \textbf{in} & \langle \eta_0, \sigma_1 \rangle
\end{array} \tag{2}$$

Now, (1) can be simply rewritten as

$$\begin{array}{lll}
[\![\mathsf{local}\ x;\ e\ \mathsf{end}]\!]\eta\sigma = & \textbf{let} & \langle \eta_0, \sigma_1 \rangle = \mathit{alloc}\ \sigma\ x \\
& \textbf{in} & [\![e]\!]\ (\eta; \eta_0)\ \sigma_1
\end{array} \tag{3}$$

Returning to our discussion of closures, suppose the expression e in such a context defines and returns a procedure, like in

$$\mathsf{local}\ x;\ \mathsf{let}\ f() = (x := \mathsf{valof}\ y)\ \mathsf{in}\ f\ \mathsf{end}$$

then the location assigned to x is built into f. Moreover, only f can access this location. The rest of the program can affect the value of the location only by calling f. It is often said that,

in such a situation, the location of x makes up the *local state* of f. More accurately, f has an exclusive "local window" on the global state (since it can access or modify the rest of the state as well). The rest of the program has neither access to, nor concerned with, the structure of this local window or the variables used for accessing it.

We will show that objects in object oriented languages can be modeled by such closures with local windows to the state. Further, the model can be extended to cover the notion of classes and inheritance as well.

We end this section with a table of the other semantic domains that we introduce in the subsequent sections. This should aid the reader as a quick reference.

$$
\begin{array}{lcll}
o & \in & \mathit{objectval} & = \mathit{menv} \\
\rho & \in & \mathit{menv} & = \mathit{message} \to \mathit{method} \\
\mu & \in & \mathit{method} & = \mathit{state} \to \mathit{val}^* \to (\mathit{val} \times \mathit{state}) \\
\xi & \in & \mathit{classval} & = \mathit{state} \to (\mathit{menv} \times \mathit{state}) \\
\psi & \in & \mathit{superclassval} & = \mathit{state} \to (\mathit{env} \times (\mathit{menv} \to \mathit{menv}) \times \mathit{state})
\end{array}
$$

---

3 ObjectTalk

The simplest of our abstract languages is *ObjectTalk*. In this language, an object can be defined using the syntax

$$e ::= \mathsf{obj}\ (x_1, \ldots, x_n)\ \{m_1(\overline{y_1}) = e_1, \ldots, m_k(\overline{y_k}) = e_k\}$$

Here $x_1, \ldots, x_n$ are the local variables of the object (also called *instance variables*), and $m_1, \ldots, m_k$ are the "messages" (or operations) that the object responds to. The definition of a message is called its "method". There is no notion of a class. However, methods can create objects each time they are called, so the effect of classes can still be achieved by objects. The syntax for sending messages to objects is

$$e ::= e_o.m(\overline{e_a})$$

where e_o is the *receiver object*, m is the message and $\overline{e_a}$ are the argument expressions. The following definition of a point object illustrates these constructs:

$$
\begin{array}{l}
p = \mathsf{obj}\ (x, y)\{ \\
\quad \begin{array}{rcl}
\mathit{put}(a, b) & = & \mathsf{begin}\ x := a;\ y := b\ \mathsf{end}, \\
\mathit{dist}() & = & \mathit{sqrt}(\mathit{sqr}(\mathsf{valof}\ x) + \mathit{sqr}(\mathsf{valof}\ y)), \\
\mathit{closer}(q) & = & \mathsf{self}.\mathit{dist}() < q.\mathit{dist}()\ \}
\end{array}
\end{array}
\tag{4}
$$

This declares two local variables x and y for the coordinates of the point, and three messages. The message *put* takes two parameters for the x and y coordinates and sets the local variables to these coordinates. The message *dist* gives the distance of the point from the origin. Finally, *closer* takes another "point-like" object q as a parameter, and checks if this point is closer to origin than q. The special variable self denotes the very object that is being defined (p, in

this case). We could have used p in place of self. But, note that p is an external name being given to the *obj* expression. We would want to define objects without giving them names. The variable self is useful to refer to the object, in such contexts.

What should objects denote? From the point of view of a user, an object simply responds to a set of messages. So, the meaning of an object should simply be an environment binding messages to their methods (message environments). The summand *objectval* of *val* can thus be defined by

$$\begin{array}{lcll} \rho & \in & objectval & = & menv = message \to method \\ \mu & \in & method & = & state \to val^* \to (val \times state) \end{array}$$

The domain *method* is similar to the type of procedure values discussed in the last section. Here is our semantics for obj-expressions:

$$\begin{array}{l} [\![\mathsf{obj}(\overline{x})\{m_i(\overline{y_i}) = e_i\}]\!]\eta\sigma = \\ \quad \textbf{let} \ \ \langle \eta_o, \sigma_1 \rangle = alloc\ \sigma\ \overline{x} \\ \qquad\qquad \text{– value environment of the object} \\ \qquad\quad \rho = \rho_\perp[m_i \to (\lambda\sigma.\ \lambda\overline{w}.\ [\![e_i]\!]\ (\eta; \eta_o[\overline{y_i} \to \overline{w}])\ \sigma)] \\ \qquad\qquad \text{– message environment} \\ \quad \textbf{in} \ \langle \rho, \sigma_1 \rangle \end{array}$$

The function *alloc* is as defined in (2) except that it is extended to deal with tuples of variables. Note that the message environment ρ produced as the value of the object expression is a closure, since the local environment η_o is completely absorbed in it. Thus the object has an exclusive window to the locations allocated in η_o.

This semantics is not yet complete because we would like to have recursive references to an object's messages in the methods defining those messages. This recursion is achieved indirectly by sending a message to the special variable self. For example, the object p defined in (4), invokes its own *dist* message in the method of *closer* using self. The use of self can be accommodated in our semantics as follows:

$$\begin{array}{l} [\![\mathsf{obj}(\overline{x})\{m_i(\overline{y_i}) = e_i\}]\!]\eta\sigma = \\ \quad \textbf{let} \ \ \langle \eta_o, \sigma_1 \rangle = alloc\ \sigma\ \overline{x} \\ \qquad\quad \rho = fix(\lambda\rho.\ \rho_\perp[m_i \to \\ \qquad\qquad\qquad (\lambda\sigma.\ \lambda\overline{w}.\ [\![e_i]\!]\ (\eta; \eta_o[\overline{y_i} \to \overline{w},\ \mathsf{self} \to \rho])\ \sigma) \\ \qquad\qquad\qquad]) \\ \quad \textbf{in} \ \langle \rho, \sigma_1 \rangle \end{array} \tag{5}$$

The only change is in the environment in which the method-expressions are interpreted. We bind the variable self to the message environment ρ that is being constructed for the object. But this makes the definition of ρ recursive, and we resolve it by introducing the fixed point operator *fix*. The use of fixed points to model references to self first appeared in [4].

The meaning of a message send is defined as follows:

$$\begin{array}{lll} [\![e_o.m(\overline{e})]\!]\eta\sigma = & \textbf{let} & \langle \rho, \sigma_1 \rangle = [\![e_o]\!]\eta\sigma \\ & & \qquad \text{– message environment of } e_o \\ & & \langle \overline{v}, \sigma_2 \rangle = [\![\overline{e}]\!]\eta\sigma_1 \\ & & \qquad \text{– values of arguments} \\ & \textbf{in} & \rho\ m\ \sigma_2\ \overline{v} \end{array} \tag{6}$$

Using the semantic definitions (5) and (6), the meaning of the point object p defined in (4) can be expressed as follows. Let α_x and α_y be two locations that can be allocated in the current state.

$$\begin{aligned}\rho_p = \mathit{fix}\,(\lambda\rho.\,[\;&\mathit{put} \rightarrow \\ &\lambda\sigma.\,\lambda\langle w_a, w_b\rangle.\,\langle w_b,\ \sigma[\alpha_x \rightarrow w_a,\ \alpha_y \rightarrow w_b]\rangle, \\ &\mathit{dist} \rightarrow \\ &\lambda\sigma.\,\lambda\langle\rangle.\,\langle\sqrt{(\sigma\,\alpha_x)^2 + (\sigma\,\alpha_y)^2},\ \sigma\rangle, \\ &\mathit{closer} \rightarrow \\ &\lambda\sigma.\,\lambda\langle\rho_q\rangle.\ \ \mathbf{let}\ \ \langle v_1, \sigma_1\rangle = \rho\,\mathit{dist}\,\sigma\,\langle\rangle \\ &\qquad\qquad\qquad\qquad \langle v_2, \sigma_2\rangle = \rho_q\,\mathit{dist}\,\sigma_1\,\langle\rangle \\ &\qquad\qquad\qquad \mathbf{in}\ \langle v_1 < v_2,\ \sigma_2\rangle\;]\,)\end{aligned}$$

Since this recursion converges finitely, we can simplify it to:

$$\begin{aligned}\rho_p = [\;&\mathit{put} \rightarrow \\ &\lambda\sigma.\,\lambda\langle w_a, w_b\rangle.\,\langle w_b,\ \sigma[\alpha_x \rightarrow w_a,\ \alpha_y \rightarrow w_b]\rangle, \\ &\mathit{dist} \rightarrow \\ &\lambda\sigma.\,\lambda\langle\rangle.\,\langle\sqrt{(\sigma\,\alpha_x)^2 + (\sigma\,\alpha_y)^2},\ \sigma\rangle, \\ &\mathit{closer} \rightarrow \\ &\lambda\sigma.\,\lambda\langle\rho_q\rangle.\ \ \mathbf{let}\ \ v_1 = \sqrt{(\sigma\,\alpha_x)^2 + (\sigma\,\alpha_y)^2} \\ &\qquad\qquad\qquad\qquad \langle v_2, \sigma_2\rangle = \rho_q\,\mathit{dist}\,\sigma\,\langle\rangle \\ &\qquad\qquad\qquad \mathbf{in}\ \langle v_1 < v_2,\ \sigma_2\rangle\;]\end{aligned} \tag{7}$$

Another idea we can think of is to let each object look at its own local state, without having a single global state that is modified by each method. Though appealing, this idea does not work. The reason is that methods can affect not only the object's local state, but also the states of objects passed as arguments. So, it is not possible to define the denotation of a method as a function of the local state alone. Instead, our semantics passes the global state to every method, but permits it to directly affect the local state only. The value environment incorporated in a method (η_o) only gives it a "window" on the local state.

4 ClassTalk

In this language, we introduce classes without inheritance. The syntax is similar to that of objects:

$$e ::= \mathsf{class}\ (x_1, \ldots, x_n)\ \{m_1(\overline{y_1}) = e_1, \ldots, m_k(\overline{y_k}) = e_k\}$$

Instance objects of classes are created by the expression

$$e ::= \mathsf{new}\ e_c$$

Now, we can define a generic point class instead of a specific point as in (4):

$$\begin{aligned}\mathit{point} = \mathsf{class}\ (x, y)\{& \\ \mathit{put}(a, b) &= \mathsf{begin}\ x := a;\ y := b\ \mathsf{end}, \\ \mathit{dist}() &= \mathit{sqrt}(\mathit{sqr}(\mathsf{valof}\ x) + \mathit{sqr}(\mathsf{valof}\ y)), \\ \mathit{closer}(q) &= \mathsf{self}.\mathit{dist}() < q.\mathit{dist}()\ \}\end{aligned} \tag{8}$$

Every evaluation of new *point* yields a new instance of *point*.

The semantics of classes should naturally satisfy the property

$$\begin{aligned}&[\![\mathsf{new\ class}(x_1,\ldots,x_n)\{m_1(\overline{y_1})=e_1,\ldots,m_k(\overline{y_k})=e_k\}]\!]\\&\quad=[\![\mathsf{obj}(x_1,\ldots,x_n)\{m_1(\overline{y_1})=e_1,\ldots,m_k(\overline{y_k})=e_k\}]\!]\end{aligned}\tag{9}$$

since instantiating a class expression to get an object is the same as directly using an *obj*-expression. So, the class construct provides an *abstraction* which can then be invoked to obtain an *objectval*. The simplest such abstraction is the domain

$$\xi \in \mathit{classval} = \mathit{state} \to (\mathit{menv} \times \mathit{state})$$

and we add it as a new summand to the *val* domain. A *classval* does not give a message environment in itself, but yields one when "instantiated" in a state. This in turn is accomplished by new.

$$\begin{aligned}&[\![\mathsf{class}(\overline{x})\{m_i(\overline{y_i})=e_i\}]\!]\eta\sigma=\\&\quad\langle\lambda\sigma'.\ \ \mathbf{let}\ \ \langle\eta_o,\sigma_1'\rangle=\mathit{alloc}\,\sigma'\,\overline{x}\\&\qquad\qquad\rho=\mathit{fix}(\lambda\rho.\,\rho_\perp[m_i\to\\&\qquad\qquad\qquad(\lambda\sigma.\,\lambda\overline{w}.\,[\![e_i]\!]\,(\eta;\eta_o[\overline{y_i}\to\overline{w},\ \mathsf{self}\to\rho])\,\sigma)\\&\qquad\qquad\qquad])\\&\qquad\mathbf{in}\ \langle\rho,\sigma_1'\rangle,\\&\quad\sigma\rangle\end{aligned}\tag{10}$$

Note that a class-expression does not change the state. It merely denotes a template for creating new objects. (Thus, we could have interpreted class expressions without reference to a state. The reason for not doing so is pedagogical. It allows us to use a single semantic function for all syntactic constructs). The meaning of new is to invoke the template:

$$\begin{aligned}[\![\mathsf{new}\ e_c]\!]\eta\sigma=\quad&\mathbf{let}\ \langle\xi,\sigma_1\rangle=[\![e_c]\!]\eta\sigma\\&\mathbf{in}\ \xi\sigma_1\end{aligned}$$

Classes do not add any expressive power to ObjectTalk owing to the equivalence (9). In fact, the effect of classes can be achieved in ObjectTalk by the following translation

$$\begin{aligned}&\mathsf{class}(\overline{x})\{\overline{M}\}\equiv\mathsf{obj}()\{new()=\mathsf{obj}(\overline{x})\{\overline{M}\}\}\\&\mathsf{new}\ c\equiv c.new()\end{aligned}$$

However, ClassTalk has an advantage from a software engineering perspective. There are good reasons to disallow free variables denoting objects in obj or class expressions. That way, we can treat every object as a self contained unit. In fact, in Smalltalk-80 no free object references are allowed in class descriptions. But, we do want class descriptions to refer to other classes. This is like importation of modules. The above simulation of classes in terms of objects does not allow such preferential treatment to free class references. So, even without inheritance, classes are useful.

The semantics we are presenting does not model the restriction that class expressions may not have free references to objects. But, it would straightforward to model the restriction by splitting the environment into a class environment and an object environment.

5 InheritTalk

In this language, we introduce a simple form of class inheritance. A subclass of another class can be expressed by the construct:

$$e ::= \textsf{subclass}\ e_c\ (x_1, \ldots, x_n)\ \{m_1(\overline{y_1}) = e_1, \ldots, m_k(\overline{y_k}) = e_k\}$$

An instance of such a subclass would have all the variables x_1, ..., x_n as well as the instance variables of the superclass e_c. Similarly, it would accept all the messages $m_1, \ldots, m_k$ as well as the messages specified in e_c. There is also a notion of overriding. That is, if a message m is specified in both the superclass and the subclass, then $o.m$ is interpreted as the method defined in the subclass. However, the behavior of instances of e_c are (reasonably) not modified by the subclass specification. This is similar to the overriding caused by statically nested scopes. In fact, our semantics of inheritance in InheritTalk closely follows that of nested scopes:

$$\begin{array}{l}
[\![\textsf{subclass}\ e_c(\overline{x})\{m_i(\overline{y_i}) = e_i\}]\!]\eta\sigma = \\
\quad \textbf{let}\ \langle \xi_c, \sigma_1 \rangle = [\![e_c]\!]\eta\sigma \\
\quad \textbf{in}\ \langle \lambda\sigma'.\ \ \textbf{let}\ \langle \rho_c, \sigma_1' \rangle = \xi_c\sigma' \\
\qquad\qquad\qquad\qquad \langle \eta_o, \sigma_2' \rangle = alloc\ \sigma_1'\ \overline{x} \\
\qquad\qquad\qquad\qquad \rho = fix(\lambda\rho.\ \rho_c[m_i \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad (\lambda\sigma.\ \lambda\overline{w}.\ [\![e_i]\!]\ (\eta; \eta_o[\overline{y_i} \rightarrow \overline{w},\ \textsf{self} \rightarrow \rho])\ \sigma) \\
\qquad\qquad\qquad\qquad\qquad\qquad]) \\
\qquad\qquad\qquad \textbf{in}\ \langle \rho, \sigma_2' \rangle, \\
\qquad\quad \sigma_1 \rangle
\end{array}$$

When instantiated in a state σ', the *classval* of the subclass first instantiates the *classval*, ξ, of the superclass. This yields a message environment ρ_c. The subclass then allocates storage for the additional instance variables $\overline{x}$, and yields the message environment ρ. This message environment is obtained by updating the environment ρ_c produced by the superclass with new message bindings for m_i. The essential difference between this and the semantics of the class construct (10) is in the use of ρ_c instead of $\rho_\perp$ in constructing ρ. The class inheritance of Simula [7] and C++ [18] work in this fashion (when virtual functions are not used).

6 SmallTalk

Note that, in InheritTalk, the variable self means different message environments in a superclass and its subclass. It can be justifiably argued that self should denote the message environment of the receiver object, and therefore should have the same meaning in both classes. Consider, for example, the following subclass *manpoint* (for Manhattan point from [10]) of the *point*

class:

$$
\begin{array}{rcl}
point = \mathsf{class}\,(x, y)\{ & & \\
\quad put(a, b) & = & \mathsf{begin}\ x := a;\ y := b\ \mathsf{end}, \\
\quad dist() & = & sqrt(sqr(\mathsf{valof}\ x) + sqr(\mathsf{valof}\ y)), \\
\quad closer(p) & = & \mathsf{self}.dist() < p.dist()\ \}
\end{array}
$$

$$
\begin{array}{l}
manpoint = \mathsf{subclass}\ point\ ()\{ \\
\quad dist() = (\mathsf{valof}\ x) + (\mathsf{valof}\ y)\}
\end{array}
$$

manpoint inherits *put* and *closer* messages from the *point* class, but uses a different notion of "distance from origin" (the sum of the x and y coordinates). We want to be able to compare *manpoints* using the *closer* operation inherited from the *point* class. But, such a use of the *closer* operation should use the *dist* method defined in *manpoint* rather than that defined in *point*. Note that InheritTalk does not achieve this kind of inheritance. What is inherited by *manpoint* in InheritTalk is a fixed behavior of an object as a *point*, as in (7). The recursion over self is already resolved in such behavior, and *closer* can only compare the Euclidean distance. But, inheritance in SMALLTALK-80 does not make such early commitment to the meaning of self. Any instance of *manpoint* consistently uses the new method for *dist* defined in the subclass definition. Similar inheritance can be achieved in C++ using "virtual" functions. We call this form of inheritance *dynamic inheritance* (and, by contrast, the inheritance of InheritTalk *static inheritance*) since the meaning of self is not determined statically by the class expression in which at appears, but dynamically when the class is instantiated.

This form of inheritance poses an interesting semantic issue. If *manpoint* inherits the "behavior" of *closer* from the *point* class, then *closer* cannot behave differently in the instances of *point* and the instances of *manpoint*. So, what is inherited from *point* is the "behavior of *closer* parameterized by the behavior of self". This means that the semantic description of a class-expression cannot directly bind self. Its binding would be known only when the class is instantiated by new. So, the meanings of class-expressions would now involve transformation *functionals* of the kind

$$\tau \in menv \to menv$$

We can think of τ as accepting the *menv* of self as a parameter, and producing a new *menv* for self. Such functionals were also involved in the semantics of ClassTalk and InheritTalk; but we could immediately eliminate them as we were only interested in the fixed points of such functionals. This we cannot do for SmallTalk.

Another semantic issue of SMALLTALK-80 that we would like to model is that the instance variables specified in a class c are visible to its subclasses. For instance, *manpoint* references the instance variables x and y specified in *point*. This means that it is not possible to hide the local environment in a class definition. These two issues motivate us to replace the subdomain *classval* of *val* by

$$\psi \in superclassval = state \to (env \times (menv \to menv) \times state)$$

The *superclassval* of a class is its meaning as seen by a subclass of it. But, to instantiate a class using new, we need its *classval*. The following mapping *close* shows that the *superclassval*

has all the information needed to determine the *classval*:

$$\begin{array}{rcl} close & : & superclassval \to classval \\ close\ \psi & = & \lambda\sigma.\ \textbf{let}\ \langle \eta, \tau, \sigma_1 \rangle = \psi\sigma \\ & & \quad \textbf{in}\ \langle fix\ \tau, \sigma_1 \rangle \end{array}$$

When instantiated in a state σ, a *superclassval* produces a triple $\langle \eta, \tau, \sigma_1 \rangle$. If the instantiation is done using *new*, then the environment η is ignored, and the fixed point of τ is produced as the object. If the instantiation is from a subclass, then the subclass can extend η with additional variables and τ with additional messages, to produce another such triple.

The syntax of SmallTalk is the same as that of InheritTalk. Only the semantics is different.

$$\begin{array}{l} [\![\mathsf{class}(\overline{x})\{m_i(\overline{y_i}) = e_i\}]\!]\eta\sigma = \\ \quad \langle \lambda\sigma'.\ \ \textbf{let}\ \ \langle \eta_o, \sigma_1' \rangle = alloc\ \sigma'\ \overline{x} \\ \qquad\qquad\qquad \eta' = \eta; \eta_o \\ \qquad\qquad\qquad \tau = \lambda\rho.\ \rho_\perp[m_i \to \\ \qquad\qquad\qquad\qquad (\lambda\sigma.\ \lambda\overline{w}.\ [\![e_i]\!]\ (\eta'[\overline{y_i} \to \overline{w},\ \mathsf{self} \to \rho])\ \sigma) \\ \qquad\qquad\qquad\qquad] \\ \qquad\qquad \textbf{in}\ \langle \eta_o, \tau, \sigma_1' \rangle, \\ \quad \sigma \rangle \end{array}$$

The major difference between this and the semantics of class-expressions in ClassTalk is that instead of producing a message environment as a fixed point of a transformation functional (τ), it produces the functional itself.

The meaning of a subclass of e_c is as follows. When instantiated in a state, it first instantiates e_c, extends the local environment created by e_c, and then extends the *menv* transformation functional determined by e_c.

$$\begin{array}{l} [\![\mathsf{subclass}\ e_c(\overline{x})\{m_i(\overline{y_i}) = e_i\}]\!]\eta\sigma = \\ \quad \textbf{let}\ \langle \psi, \sigma_1 \rangle = [\![e_c]\!]\eta\sigma \\ \quad \textbf{in}\ \ \langle \lambda\sigma'.\ \ \textbf{let}\ \ \langle \eta_c, \tau_c, \sigma_1' \rangle = \psi\sigma' \\ \qquad\qquad\qquad\qquad \langle \eta_o, \sigma_2' \rangle = alloc\ \sigma_1'\ \overline{x} \\ \qquad\qquad\qquad\qquad \eta' = \eta; \eta_c; \eta_o \\ \qquad\qquad\qquad\qquad \tau = \lambda\rho.\ \tau_c\rho[m_i \to \\ \qquad\qquad\qquad\qquad\qquad (\lambda\sigma.\ \lambda\overline{w}.\ [\![e_i]\!](\eta'[\overline{y_i} \to \overline{w},\ \mathsf{self} \to \rho])\ \sigma) \\ \qquad\qquad\qquad\qquad\qquad] \\ \qquad\qquad\qquad \textbf{in}\ \langle \eta', \tau, \sigma_2' \rangle, \\ \qquad \sigma_1 \rangle \end{array}$$

The significant part of this semantics is the definition of τ. Given a binding ρ of self, $\tau\rho$ first finds $\tau_c\rho$ (the *menv* determined by the superclass e_c) and then extends it with new message bindings. This technical meaning of SMALLTALK-80 style inheritance was independently discovered by Cook [6].

SMALLTALK-80 also has a special variable super which, appearing inside a method expression, denotes the receiver object viewed as an instance of the superclass. This can be modeled by modifying the environment used for method expressions e_i to be

$$\eta'[\overline{y_i} \to \overline{w}, \mathsf{self} \to \rho, \mathsf{super} \to \tau_c\rho]$$

The semantics of new is to close the *superclassval* to a *classval* and instantiate it in the current state:

$$[\![\textsf{new } e_c]\!]\eta\sigma = \begin{array}{l}\textbf{let } \langle \psi, \sigma_1 \rangle = [\![e_c]\!]\eta\sigma \\ \textbf{in } \mathit{close}\ \psi\ \sigma_1\end{array}$$

Let us use the *point* and *manpoint* classes to illustrate these semantic definitions. To make the meanings intuitive, we use an informal description. The *menv* transformation functionals (for an object with local environment η_o) are

$$\begin{array}{l}\tau_{point}[\eta_o] = \lambda\rho.\ [\mathit{put} \rightarrow \text{set } \eta_o x \text{ and } \eta_o y, \\ \qquad\qquad\qquad\quad \mathit{dist} \rightarrow \text{Euclidean distance}, \\ \qquad\qquad\qquad\quad \mathit{closer} \rightarrow \text{compare } \rho\,\mathit{dist} \text{ and argument's } \mathit{dist} \\ \qquad\qquad\qquad\ ] \\ \tau_{manpoint}[\eta_o] = \lambda\rho.\ [\mathit{put} \rightarrow \text{set } \eta_o x \text{ and } \eta_o y, \\ \qquad\qquad\qquad\quad \mathit{dist} \rightarrow \text{Manhattan distance}, \\ \qquad\qquad\qquad\quad \mathit{closer} \rightarrow \text{compare } \rho\,\mathit{dist} \text{ and argument's } \mathit{dist} \\ \qquad\qquad\qquad\ ]\end{array}$$

Notice that only the binding of *dist* is changed. When we *close* the *superclassvals* for instantiation, we get the respective *menvs* as fixed points:

$$\begin{array}{l}\rho_{point}[\eta_o] = [\mathit{put} \rightarrow \text{set } \eta_o x \text{ and } \eta_o y, \\ \qquad\qquad\qquad\quad \mathit{dist} \rightarrow \text{Euclidean distance}, \\ \qquad\qquad\qquad\quad \mathit{closer} \rightarrow \text{compare Euclidean distance and argument's } \mathit{dist} \\ \qquad\qquad\qquad\ ] \\ \rho_{manpoint}[\eta_o] = [\mathit{put} \rightarrow \text{set } \eta_o x \text{ and } \eta_o y, \\ \qquad\qquad\qquad\quad \mathit{dist} \rightarrow \text{Manhattan distance}, \\ \qquad\qquad\qquad\quad \mathit{closer} \rightarrow \text{compare Manhattan distance and argument's } \mathit{dist} \\ \qquad\qquad\qquad\ ]\end{array}$$

This illustrates that SMALLTALK style inheritance occurs at the *superclassval* level rather than at the *classval* level. This fact can be used for reasoning about object oriented programs as follows. When a class (or a subclass) defined, we cannot make any assumptions about the behavior of self except that it looks something like the *menv* being defined. When a class is instantiated, the instance object fixes the meaning of self and its behavior becomes determinate. Another way to think about programs is by giving two meanings to each class, in terms of *superclassvals* and *classvals*. The *superclassval* meaning is as just mentioned. The *classval* meaning assumes that self has the same behavior as the *menv* being defined. In this view, we have to remember that what is inherited is the *superclassval* and what is instantiated is the *classval*.

The fixed point involved in the above semantic definition merely models the recursion involved in references to self in class definitions. There may be another other kind of recursion involved in an object-oriented program. This involves class references rather than object references: the description of a class may create its own instances. This can be mutual recursion as well, with two classes creating each other's instances. All such recursions would be built

into the environment η which we are assuming to be available in all the semantic definitions so far. To illustrate this, we will add yet another construct to SmallTalk. This will also facilitate comparison with the semantics of Kamin who uses a similar construct:

$$e ::= \mathsf{hierarchy}\ x_1 = e_1,\ \ldots,\ x_n = e_n\ \mathsf{in}\ e$$

We intend that a construct like this be used at the top level of a program. All the expressions e_i are restricted to be class-expressions and the only free variables in them are the class names $x_1, \ldots, x_n$. These variables can then be used in the body expression e. The semantics of this is fairly conventional:

$$\begin{array}{l} [\![\mathsf{hierarchy}\ x_1 = e_1, \ldots, x_n = e_n\ \mathsf{in}\ e]\!] = \\ \quad \textbf{let}\ \ \phi = \lambda\eta.\ \eta_\perp[x_i \rightarrow fst\ ([\![e_i]\!]\eta\sigma_\perp)] \\ \qquad\ \ \eta = fix\ \phi \\ \quad \textbf{in}\ [\![e]\!]\eta\sigma_\perp \end{array}$$

We call the fixed point involved in this construction an *environmental fixed point* to distinguish it from the *fixed point over* self which we have seen before.

7 Relation to Kamin's Semantics

As mentioned in the introduction, Kamin [10] used different framework for describing the denotational semantics of SMALLTALK–80. Our work grew out of the contention that this semantic description was not sufficiently abstract. In this section, we make this observation concrete by comparing the two semantic descriptions. First of all, let us reexpress Kamin's semantics in our notation. The semantic domains are given below. We subscript the domains involved in Kamin's semantics by K. References to our semantic domains in this section are subscripted by A (for "Abstract") to distinguish them from the former.

$$\begin{array}{rcll} v, w & \in & val_K & = & basicval + loc + objectval_K + classval_K \\ \eta & \in & env_K & = & variable \rightarrow val_K \\ \sigma & \in & state_K & = & loc \rightarrow val_K \\ \rho & \in & menv_K & = & message \rightarrow method_K \\ \mu & \in & method_K & = & state_K \rightarrow objectval_K \rightarrow val_K^* \rightarrow \\ & & & & \qquad (val_K \times state_K) \\ o & \in & objectval_K & = & env_K \times classname \\ c & \in & classname & & \\ \xi & \in & classval_K & = & variable^* \times menv_K \end{array}$$

The domains $method_K$, $objectval_K$ and $classval_K$ differ from ours. The meaning of a class expression is given by:

$$\begin{aligned}
&[\![\mathsf{class}(\overline{x})\{m_i(\overline{y_i}) = e_i\}]\!]\eta\sigma = \\
&\quad \textbf{let } \rho = \rho_\perp[m_i \to \lambda\sigma.\ \lambda o_r.\ \lambda\overline{w}.\ \textbf{let } \langle \eta_r, c_r\rangle = o_r \\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in } [\![e_i]\!](\eta; \eta_r[\overline{y_i} \to \overline{w},\ \mathsf{self} \to o_r])\sigma] \\
&\quad \textbf{in } \langle(\overline{x}, \rho), \sigma\rangle
\end{aligned}$$

Let us cut through the formalism to indicate what is going on here.

1. The denotations of objects contain a *local environment* and a *classname*. The latter in turn determines a message environment. The local environment is not hidden as in our semantics. Moreover, this description involves unconventional use of the syntactic domain *classname*. Traditionally, denotational descriptions only use names as input domains of environments. But, here *classname* is being used in the semantic domain $objectval_K$. As we will shortly see, this indirect referencing of classes from *objectvals* is critical to Kamin's semantic description.

2. The denotations of classes, likewise, contain the local instance variable names in addition to the message environment. Message environments are associated with classes rather than with objects.

3. The denotations of methods take an *implicit argument* denoting the receiver object o_r, in addition to the arguments supplied in a message send. Since the denotations of methods are defined in the context of their classes, the receiver of the message is completely unknown at this point. So, it is necessary to make a method a function of the receiver. References to self in a method definition are interpreted as references to this implicit argument (Cf. the environment used for $inbracketse_i$ in the semantic definition).

4. There is a kind of *dynamic binding* involved in the interpretation of method expressions. The environment η_r is obtained dynamically as part of the parameter o_r to the method, but is used for binding the free variables in e.

The semantics of instantiation and message send clarify these points. (We first define an auxiliary function to look up the *menv* of a class in an environment):

$$\begin{aligned}
&lookup\ c\ \eta = \textbf{let } \langle \overline{x}, \rho\rangle = \eta\ c \textbf{ in } \rho \\
&[\![\mathsf{new}\ c]\!]\eta\sigma = \ \textbf{let } \ \langle \overline{x}, \rho\rangle = \eta\ c \\
&\qquad\qquad\qquad\qquad \langle \eta_o, \sigma_1\rangle = alloc\ \sigma\ \overline{x} \\
&\qquad\qquad\qquad \textbf{in } \langle\langle \eta_o, c\rangle, \sigma_1\rangle \\
&[\![e_o.m(\overline{e})]\!]\eta\sigma = \ \textbf{let } \ \langle\langle \eta_o, c_o\rangle, \sigma_1\rangle = [\![e_o]\!]\eta\sigma \\
&\qquad\qquad\qquad\qquad\qquad \rho = lookup\ c_o\ \eta \\
&\qquad\qquad\qquad\qquad\qquad \langle \overline{v}, \sigma_2\rangle = [\![\overline{e}]\!]\eta\sigma_1 \\
&\qquad\qquad\qquad\qquad \textbf{in } \rho\ m\ \sigma_2\ \langle \eta_o, c_o\rangle\ \overline{v}
\end{aligned}$$

Unlike our semantics of ClassTalk or SmallTalk, there is not yet any recursion in this semantic description. There seem to be two reasons for this. Firstly, the methods are defined as *functions*

of the receiver objects. So, there is a delayed self-reference here. Secondly, the use of *classnames* also involves a delayed self-reference. If, for instance, we directly use the environment ρ instead of *classname* c in *objectval*, then the semantics of message send would involve a self-application of ρ:

$$\rho\, m\, \sigma_2\, \langle \eta_o, \rho \rangle\, \overline{v}$$

This may then introduce implicit recursion. All these delayed recursions are pushed into the environment η. Consider again, the *point* class we have mentioned before:

$$\begin{array}{rcl} point = \mathsf{class}\,(x,y)\{\ put(a,b) & = & \mathsf{begin}\ x := a;\ y := b\ \mathsf{end}, \\ dist() & = & sqrt(sqr(\mathsf{valof}\ x) + sqr(\mathsf{valof}\ y)), \\ closer(p) & = & \mathsf{self}.dist() < p.dist()\ \} \end{array}$$

The *menv* component of this class-expression can only be determined in an environment η as follows:

$$\begin{array}{l} \rho_{point}[\eta] = [\ put \rightarrow \\ \qquad \lambda\sigma.\ \lambda\langle \eta_r, c_r \rangle.\ \lambda\langle w_a, w_b \rangle.\ \langle w_b,\ \sigma[\eta_r x \rightarrow w_a,\ \eta_r y \rightarrow w_b] \rangle, \\ \quad dist \rightarrow \\ \qquad \lambda\sigma.\ \lambda\langle \eta_r, c_r \rangle.\ \lambda\langle\rangle.\ \langle \sqrt{\sigma(\eta_r x)^2 + \sigma(\eta_r y)^2},\ \sigma \rangle, \\ \quad closer \rightarrow \\ \qquad \lambda\sigma.\ \lambda\langle \eta_r, c_r \rangle.\ \lambda\langle\langle \eta_p, c_p \rangle\rangle. \\ \qquad\quad \mathbf{let}\ \ \rho_r = lookup\, c_r\, \eta \\ \qquad\qquad\ \ \rho_p = lookup\, c_p\, \eta \\ \qquad\qquad\ \ \langle v_1, \sigma_1 \rangle = \rho_r\, dist\, \sigma\, \langle \eta_r, c_r \rangle\, \langle\rangle \\ \qquad\qquad\ \ \langle v_2, \sigma_2 \rangle = \rho_p\, dist\, \sigma_1\, \langle \eta_p, c_p \rangle\, \langle\rangle \\ \qquad\quad \mathbf{in}\ \langle v_1 < v_2,\ \sigma_2 \rangle\ ] \end{array}$$

The environment η is needed to look up the *menv* components of the classes c_r and c_p.

The delayed recursions are then captured by the environmental fixed point in the semantics of the hierarchy construct.

$$\begin{array}{l} [\![\mathsf{hierarchy}\ c_1 = e_1, \ldots, c_k = e_k\ \mathsf{in}\ e]\!] = \\ \quad \mathbf{let}\ \ \phi = \lambda\eta.\ \eta_\perp[c_i \rightarrow fst\,([\![e_i]\!]\eta\sigma_\perp)] \\ \qquad\ \ \eta = fix\ \phi \\ \quad \mathbf{in}\ [\![e]\!]\,\eta\,\sigma_\perp \end{array}$$

In [10], the functional ϕ and the environment η are given by two separate semantic functions **D** and **C** of the hierarchy. When the definition of the class *point* is given in such a hierarchy construct, the fixed point construction allows us to consider every possibility for the classes c_r

and c_p. If *point* is the only class defined, then we have

$$\begin{aligned}
\rho_{point} = [\ & put \rightarrow \\
& \lambda\sigma.\ \lambda\langle \eta_r, c_r\rangle.\ \lambda\langle w_a, w_b\rangle.\ \langle w_b,\ \sigma[\eta_r x \rightarrow w_a,\ \eta_r y \rightarrow w_b]\rangle, \\
& dist \rightarrow \\
& \lambda\sigma.\ \lambda\langle \eta_r, c_r\rangle.\ \lambda\langle\rangle.\ \langle \sqrt{\sigma(\eta_r x)^2 + \sigma(\eta_r y)^2},\ \sigma\rangle, \\
& closer \rightarrow \\
& \lambda\sigma.\ \lambda\langle \eta_r, c_r\rangle.\ \lambda\langle\langle \eta_p, c_p\rangle\rangle. \\
& \quad c_r = point \wedge c_p = point \rightarrow \\
& \qquad \textbf{let}\ \ v_1 = \sqrt{\sigma(\eta_r x)^2 + \sigma(\eta_r y)^2} \\
& \qquad\quad\ \ v_2 = \sqrt{\sigma(\eta_p x)^2 + \sigma(\eta_p y)^2} \\
& \qquad \textbf{in}\ \langle v_1 < v_2,\ \sigma\rangle; \\
& \qquad \bot\ ]
\end{aligned}$$

If the hierarchy has two other classes, say *line* and *manpoint* both of which have a message *dist*, then the binding of *closer* in ρ_{point} would define separate results for each of the 9 combinations of c_r and c_p. If *manpoint* is a subclass of *point* but *line* is unrelated, then it is certainly meaningful to handle the case c_r = *manpoint* because the receiver object may well be an instance of *manpoint*; but, the case c_r = *line* would never arise. So, passing the receiver object as an implicit argument to a method is somewhat an overkill.

The set of possibilities for the *menvs* of both the implicit and explicit arguments in this semantics is *finite* (the set of classes defined in the hierarchy construct). On the other hand, in our semantics for SmallTalk, there is only a *single* possibility for the implicit argument (determined at the time of instantiation) and the set of possibilities for the explicit arguments is *unrestricted.*

The semantics of inheritance in this framework is quite straightforward:

$$\begin{aligned}
& [\![\textsf{subclass}\ c\ (\overline{x})\{m(\overline{y}) = e\}]\!]\eta\sigma = \\
& \quad \textbf{let}\ \ \langle \overline{x}_c, \rho_c\rangle = \eta\, c \\
& \qquad \rho = \rho_c[m \rightarrow \lambda\sigma.\ \lambda o_r.\ \lambda\overline{w}.\ \textbf{let}\ \langle \eta_r, c_r\rangle = o_r \\
& \qquad\qquad\qquad\qquad\qquad\qquad \textbf{in}\ [\![e]\!](\eta; \eta_r[\overline{y} \rightarrow \overline{w},\ \textsf{self} \rightarrow o_r])\sigma] \\
& \quad \textbf{in}\ \langle\langle \overline{x} \cdot \overline{x}_c,\ \rho\rangle,\ \sigma\rangle
\end{aligned}$$

There is no need for *superclassval* because the message environment of self is obtained directly from the implicit argument denoting the receiver. Or, viewed a little differently, a *classval* in this framework is indeed like our *superclassval* because each method takes a parameter denoting self.

In summary, the denotations in Kamin's semantic description contain strictly more information than our denotations and hence are less abstract.

7.1 Classes as types

We should not, however, dismiss Kamin's framework as being just too low-level. It provides an important alternative view point of object oriented programming.

In our entirement treatment of object oriented languages, we have viewed classes merely as abstractions (procedures) used for generating instance objects. But, classes are also meant to

be *types*. That is what the very term "class" signifies. Classes are not *structural* types in the sense of [3, 13, 14], but *behavioral types* or *abstract types*. So, a semantics of classes should also throw some light on how a class may be viewed as a collection of objects and what is common to all such objects.

At the programming level (as opposed to the specification level), an abstract type is denoted by a scheme of representation and a definition of the operations on the representation. Kamin's *classvals* as pairs $\langle \overline{x}, \rho \rangle$ of instance variables and message environments precisely fit this description. What is common to all the instances of a class is that they all have the instance variables $\overline{x}$ and share the behavior ρ. Thus, Kamin's objects can merely reference their classes to determine the behvior. It is hard to state what is common among our objects with regard to their classes, since each object has its "own" behavior unrelated to other objects.

Kamin's treatment of object-oriented languages is, in fact, remarkably close to conventional abstract data type languages like Alphard [16] and CLU [8]. The operations (message environments) in these languages are associated with abstract types (classes) rather than with individual data objects. Alphard, in fact, treats the first argument of an operation as a special one for obtaining the binding of the operation name, which brings it very close to modern object oriented languages. Kamin's methods similarly use their first argument (implicit argument denoting the receiver object) as a special one for obtaining the message environment. What is different in Alphard is that the association between objects and types is determined statically. If we extend it to allow dynamic association, we would obtain a framework much like that used in Kamin's semantics of SMALLTALK-80.

These observations point to new directions for future investigation. Kamin's treatment is able to depict classes as types, but, on the negative side, makes the internal representations of objects visible in their denotations. Our treatment hides the internal representations, but loses the ability to capture the commonality of the instances of a class. Is there a way to semantically hide the representations, without losing the commonality of instances? The answer to this may lie in semantic devices like existential types [5, 15] and dependent types [2, 12].

There is another question we may ask in this connection. Kamin's semantic treatment shows that if we extend an abstract data type language like Alphard with dynamic typing and inheritance, we naturally obtain *dynamic inheritance* of the kind in our SmallTalk. Simula and C++, on the other hand, retain static typing and the inheritance obtained in them is *static inheritance* of the kind in our InheritTalk. Is there a relationship betwen static typing and static inheritance, on the one hand, and dynamic typing and dynamic inheritance, on the other?

8 Conclusion

We have described the semantics of object-oriented languages by treating objects as closures. Specifically, we interpret an object as a message environment (binding messages to methods) with a hidden local environment (binding instance variables to values or locations). We have shown that this framework can be extended to give full descriptions of classes, instantiation, and inheritance in the style of Simula as well as SMALLTALK-80.

We have also compared our approach to the semantics of SMALLTALK-80 given by Kamin in a recent paper [10]. Kamin interprets objects as pairs with local environments and class

references. Local environments are not hidden as in our semantics, and class references, rather than classes, are used in denotations. We have discussed the various implications of these differences.

Our semantics and Kamin's semantics may be seen as two different *views* of object-oriented programming. We associate operations with objects, and treat classes as abstractions (functions) used to generate such objects. Kamin associates operations with classes, and treats objects as data records with associated classes. The latter view is closer to languages based abstract data types, whereas ours is closer to conventional functional programming concepts.

9 Acknowledgements

Sam Kamin's work in formalization of the intricacies of SMALLTALK-80 was the starting point of this work. Without his formalization, this would not have been possible. I benefited greatly from discussions with him. Sam also implemented the semantics reported here, as well as his own, in ML which aided my understanding of the issues.

References

[1] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, MA, 1985.

[2] R. M. Burstall and B. Lampson. A kernel language for abstract data types and modules. In G. Kahn, D. B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, page , Springer-Verlag, 1984. Lecture Notes in Computer Science, Vol 173.

[3] L. Cardelli. A semantics of multiple inheritance. In *International symposium on semantics of data types*, pages 51–68, Springer-Verlag, 1984.

[4] L. Cardelli. A semantics of multiple inheritance. In *Semantics of Data Types*, pages 51–67, Springer-Verlag LNCS Vol. 173, 1984.

[5] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1986.

[6] W. R. Cook. *A denotational semantics of inheritance (Extended Abstract).* Technical Report, Brown University, July 1987.

[7] O.-J. Dahl and K. Nygaard. An Algol-based simulation language. *Comm. ACM*, 9(9):671–678, Sep 1966.

[8] B. Liskov et. al. *CLU Reference Manual. Lecture Notes in Computer Science Vol. 114*, Springer-Verlag, 1981.

[9] A. Goldberg and D. Robson. SMALLTALK–80*: The Language and its Implementation.* Addison-Wesley, Reading, MA, 1983.

[10] S. Kamin. Inheritance in SMALLTALK–80: a denotational definition. In *ACM Symp. on Principles of Programming Languages*, Jan 1988.

[11] R. M. Keller and G. Lindstrom. Approaching distributed database implementations through functional programming concepts. In *Intl. Conf. on Distributed Computing Systems*, IEEE, Denver, CO., May 1985.

[12] D. MacQueen. Using dependent types to express modular structure. In *ACM Symp. on Principles of Programming Languages*, pages 277–286, 1986.

[13] D. B. MacQueen and R. Sethi. A semantic model of types for applicative languages. In *Conference on LISP and Functional Programming*, pages 243–252, August 1982.

[14] R. Milner. A theory of type polymorphism in programming. *J. Computer and System Sciences*, 17:348–375, 1978.

[15] J. C. Mitchell and G. D. Plotkin. Abstract types have existential types. In *ACM Symp. on Principles of Programming Languages*, pages 37–51, Jan 1985.

[16] M. Shaw. *ALPHARD: Form and Content.* Springer-Verlag, 1981.

[17] J. E. Stoy. *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[18] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, MA, 1986.