

Two Semantic Models of Object-Oriented Languages*

Samuel N. Kamin

Uday S. Reddy

University of Illinois at Urbana-Champaign

October 4, 1993

Abstract

We present and compare two models of object-oriented languages. The first we call the *closure model* because it uses closures to encapsulate side effects on objects, and accordingly makes the operations on an object a part of that object. 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. The second we call the *data structure model* because it mimics the implementations of data structure languages like CLU in representing objects by records of instance variables, while keeping the operations on objects separate from the objects themselves. This yields a model which is very simple, at least superficially. Both the models are presented by way of a sequence of languages, culminating in a language with SMALLTALK-80-style inheritance. The mathematical relationship between them is then discussed and it is shown that the models give equivalent results. It will emerge from this discussion that more appropriate names for the two models might be the *fixed-point model* and the *self-application model*.

1 Introduction

Object-oriented languages, such as SMALLTALK-80¹ [GR83], have recently received a lot of attention. However, the term “object-oriented” does not seem to have a commonly accepted meaning. It is sometimes used to refer to the presence of *data objects with local state*, sometimes to the *coupling* of data with operations, sometimes to record *subtyping*, 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 [ASS85, KL85]. These are sometimes loosely 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 role. The second notion,

*To appear in Gunter, C. and Mitchell, J. C. (eds) *Theoretical aspects of Object-Oriented Programming*, MIT Press, 1993.

¹“SMALLTALK-80” is a trademark of ParcPlace Systems. We use here a language called “SmallTalk” (with different capitalization) as an abstraction of SMALLTALK-80.

that of coupling data with structures of operations, is used in data abstraction languages like CLU [LAB⁺81]. A distinguishing feature of these languages is that the structure of operations is shared by many data objects.

SMALLTALK and other object oriented languages, of course, go much beyond data objects with local states or coupling of data with operations. 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 or data structures is an interesting question. If so, the denotational semantics of object oriented languages can be defined in terms of such concepts.

In this paper, we present two semantic models of object-oriented languages. The closure-based model has been presented previously in [Car84, Coo89, Red88]. The data structure model appeared in [Kam88]. Reddy [Red88] also gave an abstracted version of the model in [Kam88] and commented on their relationship. The present paper is an expansion of [Red88], in which the data structure model is presented more fully and the relationship between the models is formalized. In particular, the models are shown to give equivalent results. The assertion in [Red88] that the closure model is more abstract than the data structure model is discussed, though we have not succeeded in proving it.

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, *InheritTalk*, provides subclass to be defined by inheriting from other classes. The bindings of messages used by superclasses are not affected by inheritance. The inheritance of Simula [DN66] and C++ [Str86] work in this fashion (when virtual functions are excluded). Finally, we define a language called *SmallTalk*, which implements inheritance in the style of SMALLTALK-80, by rebinding messages in subclasses.²

The four languages are defined in each of the two models in sections 3 and 4, respectively. Section 2 establishes our notations and gives the semantics of simple imperative language constructs. Section 5 comments on the relationship between the two models and, in Section 6, we formalize the relationship and prove the computational equivalence of the two models.

2 Denotational Framework

Our style of presentation will be to consider a series of small 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:

²In the terminology of P. Wegner [Weg87], *ObjectTalk* is a prototypical “object-based” language, *ClassTalk* is a “class-based” language while *InheritTalk* and *SmallTalk* are “object-oriented” languages.

$$\begin{array}{l}
x, y \in \text{variable} \\
e \in \text{expression} \\
\\
e ::= x \mid \text{valof } e \mid x := e \mid \text{let } x = e_1 \text{ in } e_2
\end{array}$$

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 [Sto77] is of the type

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

So, an expression valuation $\llbracket e \rrbracket \eta \sigma$ 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:

$$\begin{array}{l}
\alpha \in \text{loc} \\
v, w \in \text{val} = \text{basicval} + \text{loc} + \dots \\
\eta \in \text{env} = \text{variable} \dot{\rightarrow} \text{val} \\
\sigma \in \text{state} = \text{loc} \dot{\rightarrow} \text{val} \\
\\
\llbracket - \rrbracket : \text{env} \rightarrow \text{state} \rightarrow (\text{val} \times \text{state}) \\
\\
\llbracket x \rrbracket \eta \sigma = \langle \eta x, \sigma \rangle \\
\llbracket \text{valof } e \rrbracket \eta \sigma = \text{let } \langle \alpha, \sigma' \rangle = \llbracket e \rrbracket \eta \sigma \\
\quad \text{in } \alpha \in \text{loc} \rightarrow \langle \sigma' \alpha, \sigma' \rangle; ? \\
\llbracket x := e \rrbracket \eta \sigma = \text{let } \alpha = \eta x \\
\quad \langle v, \sigma_1 \rangle = \llbracket e \rrbracket \eta \sigma \\
\quad \text{in } \alpha \in \text{loc} \rightarrow \langle v, \sigma_1[\alpha \rightarrow v] \rangle; ? \\
\llbracket \text{let } x = e_1 \text{ in } e_2 \rrbracket \eta \sigma = \text{let } \langle v_1, \sigma_1 \rangle = \llbracket e_1 \rrbracket \eta \sigma \\
\quad \text{in } \llbracket e_2 \rrbracket (\eta[x \rightarrow v_1]) \sigma
\end{array}$$

Let us make a few comments about our notation. The symbol $?$ denotes an error value. We do not elaborate its meaning any further. (See [Sto77] for a detailed discussion). Environments and states are finite *partial* functions, and we often need to update them (like in the semantics of assignment above). The notation $f[x \rightarrow v]$ means a copy of the function f that maps x to v , leaving everything else unchanged. Similarly, $f[x_1 \rightarrow v_1, \dots, x_k \rightarrow v_k]$ or $f[\bar{x} \rightarrow \bar{v}]$ denotes simultaneous multiple updates. A second notational device is $f; f'$ which means the update of f with all the bindings in f' (note: f' should be a finite mapping). The symbol η_{\perp} denotes the empty environment and σ_{\perp} denotes the empty state.

Closures The notion of *closure* arises from the fact that expressions may have free (nonlocal) variables. The type $env \rightarrow state \rightarrow (val \times state)$ 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.,

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

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

$$procedure = val^* \rightarrow state \rightarrow (val \times state)$$

It can be applied to a tuple of value arguments \bar{v} and then executed in a state σ , 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

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

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*³. 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.

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:

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

Its semantics is given by

$$\begin{aligned} \llbracket \text{local } x; e \text{ end} \rrbracket \eta \sigma = & \\ \text{let } \alpha = \text{newloc } \sigma & \quad \text{– new location for } x \\ \sigma_1 = \text{extend } \sigma \alpha & \quad \text{– allocation of the location} \\ \eta_1 = \eta[x \rightarrow \alpha] & \quad \text{– local environment for } e \\ \text{in } \llbracket e \rrbracket \eta_1 \sigma_1 & \end{aligned} \tag{1}$$

newloc is the operation on states that delivers a new location; it returns that location, and *extend* returns a state in which that location has been allocated (i.e. is no longer available to *newloc*; see [Sto77, p. 287–288] for further discussion of these functions.

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

$$\begin{aligned} \text{alloc } \sigma \langle x_1, \dots, x_n \rangle = \text{let } \alpha_1 = \text{newloc } \sigma & \\ \sigma_1 = \text{extend } \sigma \alpha_1 & \\ \dots & \\ \alpha_n = \text{newloc } \sigma_{n-1} & \\ \sigma_n = \text{extend } \sigma_{n-1} \alpha_n & \\ \eta_0 = \eta_\perp[x_1 \rightarrow \alpha_1, \dots, x_n \rightarrow \alpha_n] & \\ \text{in } \langle \eta_0, \sigma_1 \rangle & \end{aligned} \tag{2}$$

³We are using the term here to denote the general notion of an abstraction with no free variables, rather than the particular representation of such abstractions as pairs of expressions and environments [Sto77, p. 44]

Now, (1) can be simply rewritten as

$$\llbracket \text{local } x; e \text{ end} \rrbracket \eta \sigma = \mathbf{let} \langle \eta_0, \sigma_1 \rangle = \text{alloc } \sigma \langle x \rangle \quad (3)$$

$$\mathbf{in} \llbracket e \rrbracket (\eta; \eta_0) \sigma_1$$

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

$$\text{local } x; \mathbf{let} f() = (x := \text{valof } y) \mathbf{in} f \text{ end} \quad (4)$$

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 interest in, 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.

Data structures The second semantic concept we use in this paper is that of a *data structure*. A data structure is a compound object whose components include data values as well as operations on those values. (Thus our use of the term “structure” draws from the mathematical notion of structure). This is in contrast to closures which only contain operations as components. Data structures are familiar from languages supporting abstract data types such as CLU [LAB⁺81] and Ada [DoD82], except that the operations in such languages are often associated with static types rather than with data values. The treatment of CLU in [Kam90] is closer to our notion of data structure.

The closure defined in (4) can be expressed as a data structure by making the local variable a part of the result:

$$\text{local } x; \mathbf{let} f(x) = (x := \text{valof } y) \mathbf{in} (x, f) \text{ end} \quad (5)$$

Note that both the location x as well as the operation on x are available to users. It may appear that data structures violate the principle of data encapsulation by offering the users direct access to representations. However, they provide flexibility in defining “privileged” users who require access to representations. We will see that inheritance as in SMALLTALK-80 involves such privileged users.

3 Closure Semantics

In this section, we develop semantics for a series of small abstract languages based on the closure model. For quick reference, we define below the various semantic domains involved in this development:

$v \in val_C$	=	$basicval + loc + objectval + classval + superclassval$
$o \in objectval$	=	$menv$
$\rho \in menv$	=	$message \rightarrow method$
$\mu \in method$	=	$val^* \rightarrow state \rightarrow (val \times state)$
$\xi \in classval$	=	$state \rightarrow (menv \times state)$
$\psi \in superclassval$	=	$state \rightarrow (env \times (menv \rightarrow menv) \times state)$

3.1 ObjectTalk

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

$$e ::= \mathbf{obj} (x_1, \dots, x_n) \{m_1(\bar{y}_1) = e_1, \dots, m_k(\bar{y}_k) = e_k\}$$

Here x_1, \dots, x_n are the local variables of the object (also called *instance variables*), and m_1, \dots, 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

$$e ::= e_o.m(\bar{e}_a)$$

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

$$\begin{aligned}
p = \mathbf{obj} (x, y) \{ & put(a, b) = \mathbf{begin} \ x := a; \ y := b \ \mathbf{end}, \\
& dist() = \mathit{sqr}(\mathit{sqr}(\mathit{valof} \ x) + \mathit{sqr}(\mathit{valof} \ y)), \\
& closer(q) = \mathbf{self}.dist() < q.dist() \}
\end{aligned} \tag{6}$$

This declares two local variables x and y for the coordinates of the point, and three messages. The *put* message sets the coordinates of the points; the *dist* message gives the distance of the point from the origin; and *closer* takes another “point-like” object q as a parameter, and checks if the current point is closer to origin than q . The special variable **self** denotes the very object 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 can simply be an environment binding messages to their methods (message environments). The summand *objectval* of *val* can thus be defined by

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

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

$$\begin{aligned} \llbracket \mathbf{obj}(\bar{x})\{m_i(\bar{y}_i) = e_i\} \rrbracket \eta\sigma = & \\ \mathbf{let} \ \langle \eta_o, \sigma_1 \rangle = alloc \ \sigma \ \bar{x} & \quad \text{-- value environment of the object} \\ \rho = \rho_{\perp}[m_i \rightarrow \lambda \bar{w}. \llbracket e_i \rrbracket (\eta; \eta_o[\bar{y}_i \rightarrow \bar{w}])] & \quad \text{-- message environment} \\ \mathbf{in} \ \langle \rho, \sigma_1 \rangle & \end{aligned}$$

Note that the message environment ρ produced as the value of the object expression is a closure, since the local environment η_o is 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**. The use of **self** is accommodated in our semantics as follows:

$$\begin{aligned} \llbracket \mathbf{obj}(\bar{x})\{m_i(\bar{y}_i) = e_i\} \rrbracket \eta\sigma = & \\ \mathbf{let} \ \langle \eta_o, \sigma_1 \rangle = alloc \ \sigma \ \bar{x} & \\ \rho = fix \ (\lambda \rho. \rho_{\perp}[m_i \rightarrow \lambda \bar{w}. \llbracket e_i \rrbracket (\eta; \eta_o[\bar{y}_i \rightarrow \bar{w}, \mathbf{self} \rightarrow \rho])]) & \quad (7) \\ \mathbf{in} \ \langle \rho, \sigma_1 \rangle & \end{aligned}$$

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 [Car84].

The meaning of a message send is defined as follows:

$$\begin{aligned} \llbracket e_o.m(\bar{e}) \rrbracket \eta\sigma = \mathbf{let} \ \langle \rho, \sigma_1 \rangle = \llbracket e_o \rrbracket \eta\sigma & \quad \text{-- message environment of } e_o \\ \langle \bar{v}, \sigma_2 \rangle = \llbracket \bar{e} \rrbracket \eta\sigma_1 & \quad \text{-- values of arguments} \\ \mathbf{in} \ \rho \ m \ \bar{v} \ \sigma_2 & \quad (8) \end{aligned}$$

Using the semantic definitions (7) and (8), the meaning of the point object p defined in (6) can be expressed as follows: (where α_x and α_y are the locations allocated for x and y)

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

Since this recursion converges finitely, it simplifies to:

$$\begin{aligned} \rho_p = [put & \rightarrow \lambda \langle w_a, w_b \rangle. \lambda \sigma. \langle w_b, \sigma[\alpha_x \rightarrow w_a, \alpha_y \rightarrow w_b] \rangle, \\ dist & \rightarrow \lambda \langle \rangle. \lambda \sigma. \langle \sqrt{(\sigma \alpha_x)^2 + (\sigma \alpha_y)^2}, \sigma \rangle, \\ closer & \rightarrow \lambda \langle \rho_q \rangle. \lambda \sigma. \mathbf{let} \ v_1 = \sqrt{(\sigma \alpha_x)^2 + (\sigma \alpha_y)^2} \\ & \quad \langle v_2, \sigma_2 \rangle = \rho_q \ dist \ \langle \rangle \ \sigma \\ & \quad \mathbf{in} \ \langle v_1 < v_2, \sigma_2 \rangle] \quad (9) \end{aligned}$$

3.2 ClassTalk

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

$$e ::= \text{class } (x_1, \dots, x_n) \{ m_1(\bar{y}_1) = e_1, \dots, m_k(\bar{y}_k) = e_k \}$$

Instance objects of classes are created by the expression

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

Now, we can define a generic point class instead of a specific point as in (6):

$$\begin{aligned} \text{point} = \text{class } (x, y) \{ & \text{put}(a, b) = \text{begin } x := a; y := b \text{ end,} \\ & \text{dist}() = \text{sqr}(\text{sqr}(\text{valof } x) + \text{sqr}(\text{valof } y)), \\ & \text{closer}(q) = \text{self.dist}() < q.\text{dist}() \} \end{aligned} \quad (10)$$

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

The semantics of classes should naturally satisfy the property

$$\begin{aligned} \llbracket \text{new class}(x_1, \dots, x_n) \{ m_1(\bar{y}_1) = e_1, \dots, m_k(\bar{y}_k) = e_k \} \rrbracket \\ = \llbracket \text{obj}(x_1, \dots, x_n) \{ m_1(\bar{y}_1) = e_1, \dots, m_k(\bar{y}_k) = e_k \} \rrbracket \end{aligned} \quad (11)$$

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

$$\xi \in \text{classval} = \text{state} \rightarrow (\text{menu} \times \text{state})$$

and we add it as a new summand to the *val* domain. The semantics of the constructs is given by:

$$\begin{aligned} \llbracket \text{class}(\bar{x}) \{ m_i(\bar{y}_i) = e_i \} \rrbracket \eta \sigma = \\ \langle \lambda \sigma'. \text{let } \langle \eta_o, \sigma'_1 \rangle = \text{alloc } \sigma' \bar{x} \\ \quad \rho = \text{fix}(\lambda \rho. \rho_{\perp} [m_i \rightarrow (\lambda \bar{w}. \lambda \sigma. \llbracket e_i \rrbracket (\eta; \eta_o [\bar{y}_i \rightarrow \bar{w}, \text{self} \rightarrow \rho]) \sigma)]) \\ \quad \text{in } \langle \rho, \sigma'_1 \rangle, \\ \quad \sigma \rangle \\ \llbracket \text{new } e_c \rrbracket \eta \sigma = \text{let } \langle \xi, \sigma_1 \rangle = \llbracket e_c \rrbracket \eta \sigma \text{ in } \xi \sigma_1 \end{aligned} \quad (12)$$

Classes do not add any expressive power to ObjectTalk owing to the equivalence (11). In fact, the effect of classes can be achieved in ObjectTalk by the following translation

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

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.

3.3 InheritTalk

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

$$e ::= \text{subclass } e_c (x_1, \dots, x_n) \{m_1(\bar{y}_1) = e_1, \dots, m_k(\bar{y}_k) = e_k\}$$

An instance of such a subclass would have all the variables x_1, \dots, x_n as well as the instance variables of the superclass e_c even though none of them would be visible to the users. Similarly, it would accept all the messages m_1, \dots, m_k as well as the messages specified in e_c . There is also a notion of overriding. That is, if a message is specified in both the superclass and the subclass, then the subclass method overrides that of the superclass. However, the behavior of the superclass instances are (reasonably) not modified by the subclass specification; only the instances of the subclass use the overriding methods. 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:

$$\begin{aligned} \llbracket \text{subclass } e_c(\bar{x})\{m_i(\bar{y}_i) = e_i\} \rrbracket \eta\sigma = & \\ \text{let } \langle \xi_c, \sigma_1 \rangle = \llbracket e_c \rrbracket \eta\sigma & \\ \text{in } \langle \lambda\sigma'. \text{let } \langle \rho_c, \sigma'_1 \rangle = \xi_c\sigma' & \\ \quad \langle \eta_o, \sigma'_2 \rangle = \text{alloc } \sigma'_1 \bar{x} & \\ \quad \rho = \text{fix}(\lambda\rho. \rho_c[m_i \rightarrow \lambda\bar{w}. \llbracket e_i \rrbracket (\eta; \eta_o[\bar{y}_i \rightarrow \bar{w}, \text{self} \rightarrow \rho])]) & \\ \quad \text{in } \langle \rho, \sigma'_2 \rangle, & \\ \sigma_1 \rangle & \end{aligned}$$

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 \bar{x} , and constructs the message environment ρ by updating ρ_c . The essential difference between this and the semantics of the `class` construct (12) is in the use of ρ_c instead of ρ_\perp in constructing ρ . The default inheritance of Simula [DN66] and C++ [Str86] work in this fashion (when virtual functions are not used).

3.4 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) of the *point* class:

$$\text{manpoint} = \text{subclass } \text{point } () \{ \text{dist}() = (\text{valof } x) + (\text{valof } y) \}$$

The class *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 (9). The recursion over

`self` is already resolved in such behavior, and `closer` can only compare the Euclidean distance. 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 it 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

$$\tau \in \text{menv} \rightarrow \text{menv}$$

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 they were immediately eliminated in favor of their fixed points.

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

$$\psi \in \text{superclassval} = \text{state} \rightarrow (\text{env} \times (\text{menv} \rightarrow \text{menv}) \times \text{state})$$

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 function `close` coerces *superclassvals* to *classvals*:

$$\begin{aligned} \text{close} & : \text{superclassval} \rightarrow \text{classval} \\ \text{close } \psi & = \lambda\sigma. \mathbf{let} \langle \eta, \tau, \sigma_1 \rangle = \psi\sigma \\ & \quad \mathbf{in} \langle \text{fix } \tau, \sigma_1 \rangle \end{aligned}$$

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 (using the above coercion `close`). But, 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 following definitions state this:

$$\begin{aligned}
\llbracket \text{class}(\bar{x})\{m_i(\bar{y}_i) = e_i\} \rrbracket \eta\sigma &= \\
\langle \lambda\sigma'. \mathbf{let} \langle \eta_o, \sigma'_1 \rangle &= \text{alloc } \sigma' \bar{x} \\
\tau &= \lambda\rho. \rho_{\perp}[m_i \rightarrow \lambda\bar{w}. \llbracket e_i \rrbracket (\eta; \eta_o[\bar{y}_i \rightarrow \bar{w}, \mathbf{self} \rightarrow \rho])] \\
\mathbf{in} \langle \eta_o, \tau, \sigma'_1 \rangle, & \\
\sigma \rangle & \\
\llbracket \text{subclass } e_c(\bar{x})\{m_i(\bar{y}_i) = e_i\} \rrbracket \eta\sigma &= \\
\mathbf{let} \langle \psi, \sigma_1 \rangle &= \llbracket e_c \rrbracket \eta\sigma \\
\mathbf{in} \langle \lambda\sigma'. \mathbf{let} \langle \eta_c, \tau_c, \sigma'_1 \rangle &= \psi\sigma' \\
\langle \eta_o, \sigma'_2 \rangle &= \text{alloc } \sigma'_1 \bar{x} \\
\tau &= \lambda\rho. \tau_{c\rho}[m_i \rightarrow \lambda\bar{w}. \llbracket e_i \rrbracket (\eta; \eta_c; \eta_o[\bar{y}_i \rightarrow \bar{w}, \mathbf{self} \rightarrow \rho])] \\
\mathbf{in} \langle \eta', \tau, \sigma'_2 \rangle, & \\
\sigma_1 \rangle &
\end{aligned}$$

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 *menu* 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 [Coo89].

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

$$\eta'[\bar{y}_i \rightarrow \bar{w}, \mathbf{self} \rightarrow \rho, \mathbf{super} \rightarrow \tau_{c\rho}]$$

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

$$\begin{aligned}
\llbracket \text{new } e_c \rrbracket \eta\sigma &= \mathbf{let} \langle \psi, \sigma_1 \rangle = \llbracket e_c \rrbracket \eta\sigma \\
&\mathbf{in} \text{close } \psi \sigma_1
\end{aligned}$$

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

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

(13)

Notice that only the binding of *dist* is changed. When we *close* the *superclassvals* for

instantiation, we get the respective *menus* as fixed points:

$$\begin{aligned}
\rho_{point}[\eta_o] &= [put \rightarrow \text{set } \eta_o x \text{ and } \eta_o y, \\
&\quad dist \rightarrow \text{Euclidean distance,} \\
&\quad closer \rightarrow \text{compare Euclidean distance and argument's } dist \quad] \\
\rho_{manpoint}[\eta_o] &= [put \rightarrow \text{set } \eta_o x \text{ and } \eta_o y, \\
&\quad dist \rightarrow \text{Manhattan distance,} \\
&\quad closer \rightarrow \text{compare Manhattan distance and argument's } dist \quad] \quad]
\end{aligned} \tag{14}$$

This illustrates that SMALLTALK-80 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 *menu* being defined. When a class is instantiated, the instance object fixes the meaning of *self* and its behavior becomes determined. 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 *menu* 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*. There may be other kinds of recursion involved in a program, *e.g.*, recursive references to classes in creating instance variables.

4 Data Structure Semantics

As discussed in the introduction, Kamin [Kam88] gave a different framework for describing the denotational semantics of SMALLTALK-80. The essential difference is that, in that framework, each object contains an explicit value environment. This corresponds to the notion of *data structure* mentioned in Section 2. Accordingly, rather than methods seeing *self* as a variable bound at object-creation time, they receive it as an argument each time they are applied. In this section, we develop the semantics of our four languages using the data structure model. Again, for quick reference, we list the semantic domains involved in this development.

$$\begin{aligned}
v \in val &= basicval + loc + objectval + classval \\
o \in objectval &= env \times menu \\
\rho \in menu &= message \rightarrow method \\
\mu \in method &= objectval \rightarrow val^* \rightarrow state \rightarrow (val \times state) \\
(\phi, \rho) \in classval &= (state \rightarrow env \times state) \times menu
\end{aligned}$$

4.1 ObjectTalk

Again, objects are no longer simply message environments, but also include a value environment for the bindings of instance variables.

$$\begin{aligned} \llbracket \text{obj}(\bar{x})\{m_i(\bar{y}_i) = e_i\} \rrbracket \eta \sigma &= \mathbf{let} \langle \eta_o, \sigma_1 \rangle = \text{alloc } \sigma \ \bar{x} \\ &\quad \rho_o = \llbracket \{m_i(\bar{y}_i) = e_i\} \rrbracket \eta \\ &\quad \mathbf{in} \langle \langle \eta_o, \rho_o \rangle, \sigma_1 \rangle \end{aligned}$$

$$\llbracket \{m_i(\bar{y}_i) = e_i\} \rrbracket \eta = [m_i \rightarrow \lambda \langle \eta_o, \rho_o \rangle. \lambda \bar{v}. \llbracket e_i \rrbracket (\eta; \eta_o[\mathbf{self} \rightarrow \langle \eta_o, \rho_o \rangle, \bar{y}_i \rightarrow \bar{v}])]$$

Each method takes an additional implicit argument, $\langle \eta_o, \rho_o \rangle$ representing self. Whenever a message is sent, its receiver is passed as an additional argument:

$$\begin{aligned} \llbracket e_o.m(\bar{e}) \rrbracket \eta \sigma &= \mathbf{let} \langle \langle \eta_o, \rho_o \rangle, \sigma_1 \rangle = \llbracket e_o \rrbracket \eta \sigma \\ &\quad \langle \bar{v}, \sigma_2 \rangle = \llbracket \bar{e} \rrbracket \eta \sigma_1 \\ &\quad \mathbf{in} \rho_o(m) \langle \eta_o, \rho_o \rangle \bar{v} \sigma_2 \end{aligned}$$

As an example, consider again the definition of a point object:

$$\begin{aligned} p = \text{obj}(x, y) \{ & \text{put}(a, b) = \mathbf{begin} \ x := a; \ y := b \ \mathbf{end}, \\ & \text{dist}() = \text{sqr}(\text{sqr}(\text{valof } x) + \text{sqr}(\text{valof } y)), \\ & \text{closer}(q) = \mathbf{self}.\text{dist}() < q.\text{dist}() \} \end{aligned}$$

Suppose α_x and α_y are the two locations that are allocated in the current state for the variables x and y when this expression is evaluated. It will produce the following object:

$$\begin{aligned} &\langle [x \rightarrow \alpha_x, y \rightarrow \alpha_y], \\ & \quad [\text{put} \rightarrow \lambda \langle \eta_r, \rho_r \rangle. \lambda \langle v_a, v_b \rangle. \lambda \sigma. \langle v_b, \sigma[\eta_r(x) \rightarrow v_a, \eta_r(y) \rightarrow v_b] \rangle, \\ & \quad \text{dist} \rightarrow \lambda \langle \eta_r, \rho_r \rangle. \lambda \langle \rangle. \lambda \sigma. \langle \sqrt{\sigma(\eta_r(x))^2 + \sigma(\eta_r(y))^2}, \sigma \rangle, \\ & \quad \text{closer} \rightarrow \lambda \langle \eta_r, \rho_r \rangle. \lambda \langle \langle \eta_a, \rho_a \rangle \rangle. \lambda \sigma. \\ & \quad \quad \mathbf{let} \langle v_1, \sigma_1 \rangle = \rho_r \ \text{dist} \ \langle \eta_r, \rho_r \rangle \ \langle \rangle \ \sigma \\ & \quad \quad \langle v_2, \sigma_2 \rangle = \rho_a \ \text{dist} \ \langle \eta_a, \rho_a \rangle \ \langle \rangle \ \sigma_1 \\ & \quad \quad \mathbf{in} \ \langle v_1 < v_2, \sigma_2 \rangle] \\ &\rangle \end{aligned} \tag{15}$$

What most clearly distinguishes this object from the one in section 3.1 is that the locations α_x and α_y do not appear in any of the methods, and the definition of *dist* is not used in the semantics of *closer*. The meanings of the methods in an object are independent of that object. In this sense, these methods are far more general than necessary, since they are equipped with the ability to be applied to any object, when in reality they will always be applied to the object in which they are contained (as can be seen in the semantics of message sends). On the other hand, this flexibility turns out to be useful for the semantics of inheritance.

4.2 ClassTalk

As in the closure model, ClassTalk does not require major changes in the semantics. As in Objecttalk, but in contrast to [Kam88], we keep the method environment defined by the class in each object. Note that the meanings of methods are not dependent on the state:

$$(\phi, \rho) \in \text{classval} = (\text{state} \rightarrow \text{env} \times \text{state}) \times \text{menv}$$

$$\llbracket \text{class}(\bar{x})\{m_i(\bar{y}_i) = e_i\} \rrbracket \eta\sigma = \langle \langle \lambda\sigma. \text{alloc } \sigma \bar{x}, \llbracket \{m_i(\bar{y}_i) = e_i\} \rrbracket \eta \rangle, \sigma \rangle$$

$$\begin{aligned} \llbracket \text{new } e \rrbracket \eta\sigma &= \mathbf{let} \langle \langle \phi, \rho \rangle, \sigma_1 \rangle = \llbracket e \rrbracket \eta\sigma \\ &\quad \langle \eta_1, \sigma_2 \rangle = \phi(\sigma_1) \\ &\quad \mathbf{in} \langle \langle \eta_1, \rho \rangle, \sigma_2 \rangle \end{aligned}$$

A *classval* has two components: an allocator for instance variables and a message environment. The former is used for each creation of an instance and the latter is directly shared by all instances.

In [Kam88], objects contained class *names*, which indirectly referred to message environments (via a separate global environment). In this reformulation, we eliminate the indirect reference and directly model *classvals* to contain the message environments.

4.3 SmallTalk

The straightforward form of inheritance in the data structure model immediately gives SMALLTALK-80 style inheritance. So, we present the semantics of SmallTalk first and postpone its adaptation to InheritTalk to the following section.

Unlike the closure semantics, the data structure model does not use a fixed point to bind self. Hence, there is no need to define a separate *superclassval* which contains functionals. To put this differently, any method is perfectly able to be sent to any object — it does not “assume” it will be sent only to an object of its own class (recall the flexibility mentioned in Section 4.1). Thus, a class inherits a method simply by copying it:

$$\begin{aligned} \llbracket \text{subclass } e_c(\bar{x}) \{m_i(\bar{y}_i) = e_i\} \rrbracket \eta\sigma &= \mathbf{let} \langle \langle \phi_c, \rho_c \rangle, \sigma' \rangle = \llbracket e_c \rrbracket \eta\sigma & (16) \\ \phi_o &= \lambda\sigma. \mathbf{let} \langle \eta_1, \sigma_1 \rangle = \phi_c \sigma \\ &\quad \langle \eta_2, \sigma_2 \rangle = \text{alloc } \sigma_1 \bar{x} \\ &\quad \mathbf{in} \langle \eta_1; \eta_2, \sigma_2 \rangle \\ \rho_o &= \rho_c; \llbracket \{m_i(\bar{y}_i) = e_i\} \rrbracket \eta \\ \mathbf{in} &\langle \langle \phi_o, \rho_o \rangle, \sigma' \rangle \end{aligned}$$

The allocator of the subclass extends the allocator of the superclass to account for the extra variables and the the message environment of the subclass is a mere extension of that of the superclass.

It hardly seems worth repeating the *point/manpoint* example here. The only difference between *point* objects, as in (15), and *manpoint* objects is that, in the the message environment of the latter, *dist* is bound to the Manhattan distance. The functions bound to *closer* are identical in both the classes. This contrasts with the closure semantics (14), where the different bindings of *dist* entail different bindings for *closer*. However, whenever a *closer* message is sent to an instance of *manpoint*, the desired behavior is still obtained because the method uses the binding of *dist* contained in the instance. Thus, *classvals* in the data structure semantics resemble the *superclassvals* of the closure semantics.

Finally, let us consider how to define “**super**” in this model. Even though the syntax of message send treats **super** as if it were an object, it is not really an object in the usual sense. For, suppose it denoted an object $o = \langle \eta_s, \rho_s \rangle$. Then, the semantics of the message send $\mathbf{super}.m(\bar{e})$ would be to invoke $\rho_s(m)$, passing o as the **self** argument. But here’s the dilemma: the message m should be “sent” to **self**, i.e., **self** should be passed as the implicit **self** argument to $\rho_s(m)$. Giving the name **super** as the receiver really only indicates where to start the method search, not who the receiver is. So, **super** really denotes a message environment – a particular view of **self** – rather than a real object. We need to define a special kind of message send to such views of **self**: (note that **super** is being overloaded as both a “reserved word” and as a variable):

$$\begin{aligned} \llbracket \mathbf{super}.m(\bar{e}) \rrbracket \eta \sigma &= \mathbf{let} \ \rho_s = \eta(\mathbf{super}) \\ &\quad o = \eta(\mathbf{self}) \\ &\quad \langle \bar{v}, \sigma_1 \rangle = \llbracket \bar{e} \rrbracket \eta \sigma \\ &\quad \mathbf{in} \ \rho_s(m) \ o \ \bar{v} \ \sigma_1 \end{aligned}$$

The variable **super** needs to be bound in the environment at the point of m ’s definition. To do this, modify the semantics of **subclass** expressions (16) so that methods have the variable **super** bound in their static environments, by letting the *env* of the class value be:

$$\rho_c; \llbracket \{m_i(\bar{y}_i) = e_i\} \rrbracket \eta[\mathbf{super} \rightarrow \rho_c]$$

Interestingly, this special treatment of **super** is not involved in the closure semantics because objects in that semantics are precisely message environments.

4.4 Inherittalk

The crucial property of Inherittalk is that when a method m defined in a class C is applied to an object o , m effectively *assumes* that o is an instance of class C and not of a subclass. Indeed, if o is an object of a subclass, as soon as it becomes the receiver of m it loses all ability to receive messages of its own class. This is because all message sends in C ’s methods are statically bound; it follows that they can only send messages defined either in C or in a superclass of C . Similarly, any instance variables of o that were declared in o ’s class cannot be accessed by m or by any method called by m with o as receiver.

Thus, inheritance can be treated by a very simple mechanism: whenever an inherited method is invoked, the receiver object is coerced up to the class in which the method is defined.

$$\begin{aligned} \llbracket \mathbf{subclass} \ e_c \ (\bar{x}) \{m_i(\bar{y}_i) = e_i\} \rrbracket \eta \sigma &= \\ \mathbf{let} \ \langle \phi_c, \rho_c \rangle, \sigma' &= \llbracket e \rrbracket \eta \sigma \\ \phi_o = \lambda \sigma. \mathbf{let} \ \langle \eta_1, \sigma_1 \rangle &= \phi_c \ \sigma \\ \langle \eta_2, \sigma_2 \rangle &= \mathit{alloc} \ \sigma_1 \ \bar{x} \\ \mathbf{in} \ \langle \eta_1; \eta_2, \sigma_2 \rangle & \\ \rho_o = (\lambda m. \lambda o. \rho(m)(\mathit{coerce} \ \bar{x} \ \rho_c \ o)) &; \llbracket \{m_i(\bar{y}_i) = e_i\} \rrbracket \eta \\ \mathbf{in} \ \langle \langle \phi_o, \rho_o \rangle, \sigma' \rangle & \end{aligned}$$

where $(\mathit{coerce} \ \bar{x} \ \rho)$ modifies an *objectval* up to its superclass:

$$\begin{aligned} \mathit{coerce} \ \bar{x} \ \rho_c &: \mathit{objectval} \rightarrow \mathit{objectval} \\ &: \langle \eta, \rho \rangle \mapsto \langle \eta - \bar{x}, \rho_c \rangle \end{aligned}$$

5 Relationship between the semantic models

The most obvious relationship between these two models is that they give the same results (if one were an operational semantics and the other denotational, this would be called *adequacy* [Gun92]). This is proven in section 6.

A deeper relationship is that, as observed in [Red88] and [CP93], the closure model is more abstract than the data structure model. Unfortunately, we do not yet have a formal proof of this, but we have good reasons to believe it is so, and in the rest of this section we present those reasons.

There are two ways in which the closure model (which we refer to as the \mathcal{C} -model in this section) is more abstract than the data structure model (the \mathcal{D} -model). The more obvious one is that, by including an explicit value environment in each object, the \mathcal{D} -model fails to abstract away from instance variable names. The less obvious is that the two models rely on different models of recursion: an explicit fixed point in the \mathcal{C} -model vs. self-application in the \mathcal{D} -model. The use of self-application essentially entails that \mathcal{D} -objects contain functionals, while the \mathcal{C} -objects contain their fixed points. Since there are “more” functionals than fixed points, there are more \mathcal{D} -objects than \mathcal{C} -objects. In this section, we explain how these two models of recursion are employed.

First, let us briefly examine the first source of non-abstractness in \mathcal{D} , the presence of value environments in objects. It is easy to see that a kind of α -equivalence applies to object definitions in \mathcal{C} :

$$\mathcal{C}[\mathbf{obj}(\bar{x})\{m_i(\bar{y}_i) = e_i\}] \equiv \mathcal{C}[\mathbf{obj}(\bar{z})\{m_i(\bar{y}_i) = e_i[\bar{z}/\bar{x}]\}]$$

as long as the usual name conflicts are avoided. This equivalence fails to hold in \mathcal{D} . The semantics of \mathbf{obj} -expressions exports the value environment to outside and then receives it back as part of the receiver arguments of methods. Thus, even though renaming is not observationally distinguishable, the equivalence is not present in the semantic values themselves.

On the other hand, this rule does not extend to `class` or `subclass` expressions in SmallTalk. Since a subclass sees the instance variables of its superclass, those variables are part of the superclass’s meaning.

We can now focus on the other source of non-abstractness in the \mathcal{D} -model: it and the \mathcal{C} -model are based on distinct models of recursion. The former uses self-application familiar from (untyped) lambda calculus while the latter uses explicit fixed points.

To get to the heart of the matter, we use a functional version of ObjectTalk (i.e., without states or locations). We also assume that objects have no instance variables and that all messages take a single argument. Further, we represent message environments as tuples of methods rather than as maps from message names to methods. All these simplifications, except for the one regarding states, are essentially benign. Since the functional ObjectTalk has all the expressive power of untyped lambda calculus, the other features can be defined within the simplified language. The semantic domains for the language are

$$\begin{aligned} val_{\mathcal{C}} &= basicval + objectval_{\mathcal{C}} \\ objectval_{\mathcal{C}} &= [val_{\mathcal{C}} \rightarrow val_{\mathcal{C}}]^* \\ val_{\mathcal{D}} &= basicval + objectval_{\mathcal{D}} \\ objectval_{\mathcal{D}} &= [objectval_{\mathcal{D}} \rightarrow val_{\mathcal{D}} \rightarrow val_{\mathcal{D}}]^* \end{aligned} \tag{17}$$

The semantic functions for object definitions and message sends are as follows

$$\begin{aligned}
\mathcal{C}[\mathbf{obj}(\{m_i(x) = e_i\}_{i=1,n})]\eta &= \mathit{fix} (\lambda\rho. \langle \lambda v. \mathcal{C}[\![e_i]\!] \eta[\mathbf{self} \rightarrow \rho, x \rightarrow v] \rangle_{i=1,n}) \\
\mathcal{C}[e_0.m_i(e_1)]\eta &= (\mathcal{C}[\![e_0]\!] \eta)_i (\mathcal{C}[\![e_1]\!] \eta) \\
\mathcal{D}[\mathbf{obj}(\{m_i(x) = e_i\}_{i=1,n})]\eta &= \langle \lambda\rho. \lambda v. \mathcal{D}[\![e_i]\!] \eta[\mathbf{self} \rightarrow \rho, x \rightarrow v] \rangle_{i=1,n} \\
\mathcal{D}[e_0.m_i(e_1)]\eta &= (\mathcal{D}[\![e_0]\!] \eta)_i (\mathcal{D}[\![e_1]\!] \eta)
\end{aligned}$$

Now, consider an object with a single message m :

$$E \equiv \mathbf{obj}(\{m(x) = e\})$$

Assume that all the uses of \mathbf{self} in e are message sends of the form $\mathbf{self}.m(e')$. The \mathcal{C} -semantics of such an object is of the form $\mathit{fix} t_c$ where t_c is a functional of type $[val_C \rightarrow val_C] \rightarrow [val_C \rightarrow val_C]$. The \mathcal{D} -semantics of the object is a function t_d of type $[objectval_D \rightarrow val_D \rightarrow val_D] \rightarrow [val_D \rightarrow val_D]$. The meaning of each message send to \mathbf{self} is of the form $\rho \rho v$ where ρ is the argument of t_d . So, there is a function $t'_d : [val_D \rightarrow val_D] \rightarrow [val_D \rightarrow val_D]$ such that

$$t'_d(\rho \rho) = t_d(\rho)$$

The meaning of the object in the two semantics is

$$\begin{aligned}
o_c &= \mathit{fix} t_c \\
o_d &= \lambda\rho. t'_d(\rho \rho)
\end{aligned}$$

Modulo the fact that the two semantics work in different domains, t_c and t'_d are essentially equivalent. They both express the meaning of e as a function of what can be written as $\mathbf{self}.m$ and the formal parameter x . (“ $\mathbf{self}.m$ ” is the $[val \rightarrow val]$ function obtained by invoking the message. It is simply ρ in the \mathcal{C} -semantics, and $\rho \rho$ in the \mathcal{D} -semantics). Finally, consider a use of the object, such as

$$E.m(e')$$

The function denoted by $E.m$ in the \mathcal{C} -semantics is $o_c = \mathit{fix} t_c$, and the function denoted in the \mathcal{D} -semantics is

$$o_d o_d = (\lambda\rho. t'_d(\rho \rho)) (\lambda\rho. t'_d(\rho \rho))$$

The latter is $\mathbf{Y} t'_d$ for the familiar \mathbf{Y} combinator of lambda calculus which is equivalent to the fixed point operator [Par70, Wad76]. Thus, both the closure semantics and the data structure semantics express the same meaning by different means.

In Fig. 1 these ideas are illustrated for three examples: the simple case just described of an object with one message; an object with two messages; and an object that inherits a method.⁴ In each case, it may be verified that the functions denoting the messages have the same infinite expansions.

Having seen how the two models of recursion are employed, it is clear how the \mathcal{C} -model is the more abstract: there is, roughly speaking, a function from \mathcal{D} -meanings of objects to their

⁴If $f : X \rightarrow A$ and $g : Y \rightarrow B$, we use the notation $f \times g$ for the function $\lambda\langle x, y \rangle. \langle fx, gy \rangle$ of type $X \times Y \rightarrow A \times B$. If $f : X \rightarrow A$ and $g : X \rightarrow B$, we use the notation $[f, g]$ for the function $\lambda x. \langle fx, gx \rangle$ of type $X \rightarrow A \times B$.

Closure semantics

An object with a single message:

let $m = fix\ t$
in ... m ...

An object with two messages:

let $\langle m_1, m_2 \rangle = fix\ (t_1 \times t_2)$
in ... m_1 ...

An object inheriting a method from another:

let $\langle m_{11}, m_{12} \rangle = fix\ (t_1 \times t_{12})$
 $\langle m_{21}, m_{22} \rangle = fix\ (t_1 \times t_{22})$
in ... m_{21} ...

Data abstraction semantics

let $m' = \lambda s. t'(s\ s)$
in ... $(m\ m)$...

let $\langle m'_1, m'_2 \rangle = \langle \lambda \langle x_1, x_2 \rangle. t'_1\ ([x_1, x_2]\ \langle x_1, x_2 \rangle),$
 $\lambda \langle x_1, x_2 \rangle. t'_2\ ([x_1, x_2]\ \langle x_1, x_2 \rangle) \rangle,$
in ... $(m'_1\ \langle m'_1, m'_2 \rangle)$...

let $\langle m'_{11}, m'_{12} \rangle = \langle \lambda \langle x_1, x_2 \rangle. t'_1\ ([x_1, x_2]\ \langle x_1, x_2 \rangle),$
 $\lambda \langle x_1, x_2 \rangle. t'_{12}\ ([x_1, x_2]\ \langle x_1, x_2 \rangle) \rangle,$
 $\langle m'_{21}, m'_{22} \rangle = \langle m'_{11},$
 $\lambda \langle x_1, x_2 \rangle. t'_{22}\ ([x_1, x_2]\ \langle x_1, x_2 \rangle) \rangle$
in ... $(m'_{21}\ \langle m'_{21}, m'_{22} \rangle)$...

Figure 1: Interpretations of simple example objects

\mathcal{C} -meanings, namely $\mathcal{C} \rightarrow \mathcal{D}(o_d) = o_d o_D$. The discussion above also is a preview of the proof of equivalence of the two semantics. As mentioned before, the simplifications made to the syntax are benign and they can be easily removed while retaining the spirit of the proof. However, the assumption about the restricted use of **self** (all the uses of **self** in a method are message sends) is a serious restriction. In the next section, we formally prove that the two semantics give the same computed results for all expressions.

6 Equivalence

In this section, we formally prove that the data structure semantics and the closure semantics are equivalent for functional ObjectTalk. For this theoretical study, we use a simplified version of ObjectTalk where objects have no instance variables and support a single message. This language is defined by the abstract syntax:

$$e ::= x \mid k \mid \mathbf{obj}\{\lambda x. e'\} \mid e_0(e_1)$$

where x ranges over variables, k over constants, **obj**-expressions define objects with a single message $\lambda x. e'$, and $e_0(e_1)$ denotes the invocation of the unique message of e_0 with the argument e_1 .

The domain of the closure semantics is⁵

$$C = B + [C \rightarrow C]$$

⁵In this section, we abbreviate *basicval* to B , *val_C* to C and *val_D* to D .

An object here is denoted by a function for its unique message. For simplicity, we assume that all constants denote atomic values and there is a function $Const$ mapping constant symbols to their denotations in B . The semantics of the constructs is specified by:

$$\begin{aligned} \mathcal{C}[[x]]\eta &= \eta x \\ \mathcal{C}[[k]]\eta &= Const(k) \\ \mathcal{C}[[\text{obj}\{\lambda x.e'\}]]\eta &= \text{fix}(\lambda o.\lambda v. \mathcal{C}[[e']]\eta[\text{self}\rightarrow o, x\rightarrow v]) \\ \mathcal{C}[[e_0(e_1)]]\eta &= \begin{cases} \perp, & \text{if } \mathcal{C}[[e_0]]\eta \text{ is } \perp \text{ or is in } B \\ \mathcal{C}[[e_0]]\eta \mathcal{C}[[e_1]]\eta, & \text{otherwise} \end{cases} \end{aligned}$$

Note that the meaning of **obj**-expressions involves the fixed point operator and message send is simply function application.

The domain of the data structure semantics is simply

$$D = B + [D \rightarrow D \rightarrow D]$$

Objects here represent functions of two arguments, one for **self** and the second for the explicit argument. (The domain is slightly larger than in (17) in that the first argument is not restricted to $objectval_D$). The semantics of the language is then specified as

$$\begin{aligned} \mathcal{D}[[x]]\eta &= \eta x \\ \mathcal{D}[[k]]\eta &= Const(k) \\ \mathcal{D}[[\text{obj}\{\lambda x.e'\}]]\eta &= \lambda o.\lambda v. \mathcal{D}[[e']]\eta[\text{self}\rightarrow o, x\rightarrow v] \\ \mathcal{D}[[e_0(e_1)]]\eta &= \begin{cases} \perp, & \text{if } \mathcal{D}[[e_0]]\eta \text{ is } \perp \text{ or is in } B \\ \mathcal{D}[[e_0]]\eta \bullet \mathcal{D}[[e_1]]\eta, & \text{otherwise} \end{cases} \end{aligned}$$

The binary operator \bullet corresponding to message send is defined by $f \bullet x = f f x$. Note that it involves a self-application of the object.

The most direct way to establish a relationship between the two semantics would be to exhibit a relation θ between D and C which is respected by the two semantics. The relation should be the identity on B and two objects must be related if, given θ -related arguments, they return θ -related results, i.e., $\theta(o_d, o_c)$ iff, for all v_d and v_c , $\theta(v_d, v_c) \Rightarrow \theta(o_d o_d v_d, o_c v_c)$. Unfortunately, such a relation does not seem to exist because of the self-application o_d . The standard techniques for establishing such relations [Mil74, MS76, Rey74] fail. Therefore, we follow a circuitous approach and show that each semantics approximates the other. The next two subsections are devoted to this problem.

6.1 Data structure semantics approximates closure semantics

Since the data structure semantics models message send by self-application, the equivalence proof mirrors the inverse limit construction for the domain D . Specifically, the domain is the inverse limit D_∞ of the sequence $D_0 \xleftrightarrow{\leftarrow} D_1 \xleftrightarrow{\leftarrow} D_2 \xleftrightarrow{\leftarrow} \dots$ where $D_0 = \{\perp\}$ and each D_i is a retraction of D_{i+1} as determined by the functor

$$T(D) = B + [D \rightarrow D \rightarrow D]$$

The details of the construction may be found in [Rey72, SP82, Wad76]. For our purposes, it is adequate to note that each D_i has an isomorphic image in D_∞ and there is a retraction P_i from D_∞ to this image. The objects in D_i (and, hence, those in the isomorphic image of D_i in D_∞) are able to perform message sends at most $i - 1$ levels deep, either to themselves or to other objects.

To model this state of affairs, we extend the language as follows:

$$e ::= x \mid k \mid \mathbf{obj}\{\lambda x.e'\} \mid e_0(e_1) \mid \mathbf{obj}^i\{\lambda x.e'\}$$

The interpretation of the new expressions is

$$\begin{aligned} \mathcal{D}[\mathbf{obj}^i\{\lambda x.e'\}]\eta &= P_i(\mathcal{D}[\mathbf{obj}\{\lambda x.e'\}]\eta) \\ \mathcal{C}[\mathbf{obj}^i\{\lambda x.e'\}]\eta &= \mathcal{C}[\mathbf{obj}\{\lambda x.e'\}]\eta \end{aligned}$$

That is, in the data structure semantics, the labeled expressions denote approximate objects but, in the closure semantics, the labels have no effect. An expression in which all \mathbf{obj} -expressions have labels is called an *approximate expression*.

Lemma 1 $\mathcal{D}[\mathbf{obj}\{\lambda x.e'\}] = \sqcup_i \mathcal{D}[\mathbf{obj}^i\{\lambda x.e'\}]$

That is, every expression in the original language is expressible as the lub of approximate expressions in the extended language. The proof uses the fact that $\sqcup_i P_i = id$.

We would like to prove that the meanings assigned by the data structure semantics are “smaller” than those assigned by the closure semantics. However, this cannot be simply stated as $\mathcal{D}[e]\eta \sqsubseteq \mathcal{C}[e]\eta$ because, for higher-order values, the two meanings do not even have the same type. So, we should be satisfied if all the observable atomic values obtained in the data structure semantics are smaller than the corresponding atomic values obtained in the closure semantics. The following is the most natural requirement:

Theorem 2 For every $c \in C$, there exists a relation over D denoted by $d \lesssim c$ such that $d \lesssim c$ if and only if

1. $d = \perp$,
2. $d \in B$, $c \in B$, and $d \sqsubseteq c$, or
3. $d \in [D \rightarrow D \rightarrow D]$, $c \in [C \rightarrow C]$, and, for all d' and c' , $d' \lesssim c' \Rightarrow d \bullet d' \lesssim cc'$.

The existence of such a relation is established using the standard techniques [Mil74, MS76, MP87, Rey74]. See Appendix for the technical details. It may also be verified by induction on n that:

Lemma 3 For $d \in D$ and $c \in C$, $d \lesssim c$ if and only if, whenever $d_1 \lesssim c_1, \dots, d_n \lesssim c_n$ (for $n \geq 0$), $d \bullet d_1 \bullet \dots \bullet d_n \in B \Rightarrow d \bullet d_1 \bullet \dots \bullet d_n \sqsubseteq c c_1 \dots c_n$.

The advantage of this restatement of \lesssim is that it does not have recursive references to \lesssim in the consequent. Now, we are able to present the main result:

Theorem 4 If η_d and η_c are environments such that $\eta_d x \lesssim \eta_c x$ for all x , then $\mathcal{D}[e]\eta_d \lesssim \mathcal{C}[e]\eta_c$ for all expressions e .

Proof: We first prove the statement for *approximate* expressions in the extended language (i.e., all *obj*-expressions have integer labels). The proof is by structural induction on expressions and the integer labels on the *obj*-expressions.

- $e = x_i$. The result follows from the assumption on η_d and η_c .
- $e = k$. The result is trivial because both the semantics assign the meaning $Const(k)$.
- $e = e_0(e_1)$. By inductive hypothesis, $\mathcal{D}[[e_0]]\eta_d \lesssim \mathcal{C}[[e_0]]\eta_c$. If the former is \perp or is in B , $\mathcal{D}[[e_0(e_1)]]\eta_d = \perp$ and the result is trivial. Assume it is in $[D \rightarrow D \rightarrow D]$. Then, by definition of \lesssim , $\mathcal{C}[[e_0]]\eta_c$ is also in $[C \rightarrow C]$ and, whenever $d' \lesssim c'$, $\mathcal{D}[[e_0]]\eta_d \bullet d' \lesssim \mathcal{C}[[e_0]]\eta_c c'$. So, using the inductive hypothesis for e_1 ,

$$\mathcal{D}[[e_0]]\eta_d \bullet \mathcal{D}[[e_1]]\eta_d \lesssim \mathcal{C}[[e_0]]\eta_c \mathcal{C}[[e_1]]\eta_c$$

- $e = \mathbf{obj}^0\{\lambda x.e'\}$. $\mathcal{D}[[e]]\eta_d = P_0(\mathcal{D}[\mathbf{obj}\{\lambda x.e'\}]) = \perp \lesssim \mathcal{C}[[e]]\eta_c$.
- $e = \mathbf{obj}^i\{\lambda x.e'\}$ and $i > 0$. To show that $\mathcal{D}[[e]]\eta_d \lesssim \mathcal{C}[[e]]\eta_c$ we use Lemma 6.1. Assume that $d_1 \lesssim c_1, \dots, d_n \lesssim c_n$ and $\mathcal{D}[[e]]\eta_d \bullet d_1 \bullet \dots \bullet d_n \in B$. The objective is to show that $\mathcal{D}[[e]]\eta_d \bullet d_1 \bullet \dots \bullet d_n \sqsubseteq \mathcal{C}[[e]]\eta_c c_1 \dots c_n$. If $n = 0$, $\mathcal{D}[[e]]\eta_d \notin B$, contradicting the assumption. Assume that $n > 0$. First, calculate

$$\begin{aligned} \mathcal{D}[[e]]\eta_d &= P_i(\mathcal{D}[\mathbf{obj}\{\lambda x.e'\}]\eta_d) \\ &= P_i(\lambda o.\lambda v.\mathcal{D}[[e']]\eta_d[\mathbf{self} \rightarrow o, x \rightarrow v]) \\ &= \lambda o.\lambda v. P_{i-1}(\mathcal{D}[[e']]\eta_d[\mathbf{self} \rightarrow P_{i-1}(o), x \rightarrow P_{i-1}(v)]) \end{aligned}$$

Next, notice that

$$P_{i-1}(\mathcal{D}[[e]]\eta_d) = P_{i-1}(P_i(\mathcal{D}[\mathbf{obj}\{\lambda x.e'\}]\eta_d)) = P_{i-1}(\mathcal{D}[\mathbf{obj}\{\lambda x.e'\}]\eta_d) = \mathcal{D}[\mathbf{obj}^{i-1}\{\lambda x.e'\}]\eta_d$$

So,

$$\begin{aligned} \mathcal{D}[[e]]\eta_d \bullet d_1 \bullet \dots \bullet d_n &= (\lambda o.\lambda v. P_{i-1}(\mathcal{D}[[e']]\eta_d[\mathbf{self} \rightarrow P_{i-1}(o), x \rightarrow P_{i-1}(v)])) \bullet d_1 \bullet \dots \bullet d_n \\ &= P_{i-1}(\mathcal{D}[[e']]\eta_d[\mathbf{self} \rightarrow P_{i-1}(\mathcal{D}[[e]]\eta_d), x \rightarrow P_{i-1}(d_1)]) \bullet d_2 \bullet \dots \bullet d_n \\ &\sqsubseteq \mathcal{D}[[e']]\eta_d[\mathbf{self} \rightarrow \mathcal{D}[\mathbf{obj}^{i-1}\{\lambda x.e'\}]\eta_d, x \rightarrow d_1] \bullet d_2 \bullet \dots \bullet d_n \end{aligned}$$

Similarly,

$$\begin{aligned} \mathcal{C}[[e]]\eta_c c_1 \dots c_n &= \mathit{fix}(\lambda o.\lambda v.\mathcal{C}[[e']]\eta_c[\mathbf{self} \rightarrow o, x \rightarrow v]) c_1 \dots c_n \\ &= \lambda v.\mathcal{C}[[e']]\eta_c[\mathbf{self} \rightarrow \mathcal{C}[[e]]\eta_c, x \rightarrow v] c_1 \dots c_n \\ &= \mathcal{C}[[e']]\eta_c[\mathbf{self} \rightarrow \mathcal{C}[[e]]\eta_c, x \rightarrow c_1] c_2 \dots c_n \end{aligned}$$

Now, by inductive hypothesis (for the smaller label $i - 1$),

$$\mathcal{D}[\mathbf{obj}^{i-1}\{\lambda x.e'\}]\eta_d \lesssim \mathcal{C}[\mathbf{obj}^{i-1}\{\lambda x.e'\}]\eta_c = \mathcal{C}[[e]]\eta_c$$

Since $d_1 \lesssim c_1$ by assumption, the inductive hypothesis applied to e' gives

$$\mathcal{D}[[e']]\eta_d[\mathbf{self} \rightarrow \mathcal{D}[\mathbf{obj}^{i-1}\{\lambda x.e'\}]\eta_d, x \rightarrow d_1] \lesssim \mathcal{C}[[e']]\eta_c[\mathbf{self} \rightarrow \mathcal{C}[[e]]\eta_c, x \rightarrow c_1]$$

Hence, by Lemma 6.1,

$$\mathcal{D}[[e']]_{\eta_d}[\text{self} \rightarrow \mathcal{D}[\text{obj}^{i-1}\{\lambda x.e'\}]_{\eta_d}, x \rightarrow d_1] \bullet d_2 \bullet \dots \bullet d_n \sqsubseteq \mathcal{C}[[e']]_{\eta_c}[\text{self} \rightarrow \mathcal{C}[[e]]_{\eta_c}, x \rightarrow c_1] c_2 \dots c_n$$

Thus,

$$\mathcal{D}[[e]]_{\eta_d} \bullet d_1 \bullet \dots \bullet d_n \sqsubseteq \mathcal{C}[[e]]_{\eta_c} c_1 \dots c_n$$

This shows the result for all the approximate expressions.

Next, consider the full extended language. The only interesting case is $e = \text{obj}\{\lambda x.e'\}$. (The other cases are similar to the above treatment). We again use Lemma 6.1 and assume that $d_i \lesssim c_i$ for $i = 1, \dots, n$, and $\mathcal{D}[[e]]_{\eta_d} \bullet d_1 \bullet \dots \bullet d_n \in B$. By Lemma 6.1, $\mathcal{D}[[e]]_{\eta_d}$ is equal to $\bigsqcup_i \mathcal{D}[\text{obj}^i\{\lambda x.e'\}]_{\eta_d}$, and, since \bullet is continuous,

$$\mathcal{D}[[e]]_{\eta_d} \bullet d_1 \bullet \dots \bullet d_n = \bigsqcup_i (\mathcal{D}[\text{obj}^i\{\lambda x.e'\}]_{\eta_d} \bullet d_1 \bullet \dots \bullet d_n)$$

Use the same reasoning as above to conclude that

$$\mathcal{D}[\text{obj}^i\{\lambda x.e'\}]_{\eta_d} \bullet d_1 \bullet \dots \bullet d_n \sqsubseteq \mathcal{C}[\text{obj}\{\lambda x.e'\}]_{\eta_c} c_1 \dots c_n$$

and the lub of the LHS's is smaller than the right hand side.

Since the result holds for the extended language, it must hold for the subset representing the original language as well. ■

6.2 Closure semantics approximates data structure semantics

This proof follows the same line of argument as the previous one. It is slightly simpler because there is no self-application involved in message send.

Theorem 5 For every $d \in D$, there exists a relation over C denoted by $c \lesssim d$ such that $c \lesssim d$ if and only if

1. $c = \perp$,
2. $c \in B$, $d \in B$ and $c \sqsubseteq d$,
3. $c \in [C \rightarrow C]$, $d \in [D \rightarrow D \rightarrow D]$ and, for all c' and d' , $c' \lesssim d' \Rightarrow c c' \lesssim d \bullet d'$.

This relation is constructed by induction on C_∞ construction. We also have:

Lemma 6 For $c \in C$ and $d \in D$, $c \lesssim d$ if and only if, whenever $c_1 \lesssim d_1, \dots, c_n \lesssim d_n$ (for $n \geq 0$), $c c_1 \dots c_n \in B \Rightarrow c c_1 \dots c_n \sqsubseteq d \bullet d_1 \bullet \dots \bullet d_n$.

Theorem 7 If η_c, η_d are environments such that $\eta_c x \lesssim \eta_d x$ for all x , then $\mathcal{C}[[e]]_{\eta_c} \lesssim \mathcal{D}[[e]]_{\eta_d}$ for all expressions e .

Proof: The proof is by structural induction on expressions.

- $e = x$. Follows by assumption on η_c and η_d .

- $e = k$. Both the semantics assign $Const(k)$.
- $e = e_0(e_1)$. By inductive hypothesis, $\mathcal{C}[[e_0]]\eta_c \lesssim \mathcal{D}[[e_0]]\eta_d$. If $\mathcal{C}[[e_0]]\eta_c$ is \perp or is in B , $\mathcal{C}[[e]]\eta_c = \perp$ and the result is trivial. If $\mathcal{C}[[e_0]]\eta_c$ is in $[C \rightarrow C]$ then, by definition, $\mathcal{D}[[e_0]]\eta_d$ is in $[D \rightarrow D \rightarrow D]$ and, whenever $c' \lesssim d'$, $\mathcal{C}[[e_0]]\eta_{c'} \lesssim \mathcal{D}[[e_0]]\eta_{d'} \bullet d'$. Since $\mathcal{C}[[e_1]]\eta_c \lesssim \mathcal{D}[[e_1]]\eta_d$ by inductive hypothesis, the result follows.
- $e = \text{obj}\{\lambda x.e'\}$. First, calculate

$$\begin{aligned} & \mathcal{C}[[\text{obj}\{\lambda x.e'\}]]\eta_c \\ &= \text{fix}(\lambda o.\lambda v.\mathcal{C}[[e']] \eta_c[\text{self} \mapsto o, x \mapsto v]) \\ &= \bigsqcup_i (\lambda o.\lambda v.\mathcal{C}[[e']] \eta_c[\text{self} \mapsto o, x \mapsto v])^i(\perp) \end{aligned}$$

Call the functional involved here F , so that the semantics is $\bigsqcup_i F^i(\perp)$. Since \lesssim is inclusive in its C argument, this holds if $F^i(\perp) \lesssim \mathcal{D}[[e]]\eta_d$. We show the latter by induction on i .

For $i = 0$, $F^0(\perp) = \perp \lesssim \mathcal{D}[[e]]\eta_d$. Next, consider $i > 0$. To use Lemma 6.2, assume that $c_1, \dots, c_n \in C$ and $d_1, \dots, d_n \in D$ are such that $c_i \lesssim d_i$ and $c \ c_1 \dots c_n \in B$. (So, $n > 0$). Note that $F^i(\perp) = F(F^{i-1}(\perp)) = \lambda v.\mathcal{C}[[e']] \eta_c[\text{self} \mapsto F^{i-1}(\perp), x \mapsto v]$. Hence,

$$F^i(\perp) \ c_1 \dots c_n = \mathcal{C}[[e']] \eta_c[\text{self} \mapsto F^{i-1}(\perp), x \mapsto c_1] \ c_2 \dots c_n$$

Similarly,

$$\mathcal{D}[[e]]\eta_d \bullet d_1 \bullet \dots \bullet d_n = \mathcal{D}[[e']] \eta_d[\text{self} \mapsto \mathcal{D}[[e]]\eta_d, x \mapsto d_1] \bullet d_2 \bullet \dots \bullet d_n$$

Since

$$\mathcal{C}[[e']] \eta_c[\text{self} \mapsto F^{i-1}(\perp), x \mapsto c_1] \lesssim \mathcal{D}[[e']] \eta_d[\text{self} \mapsto \mathcal{D}[[e]]\eta_d, x \mapsto d_1]$$

(by inductive hypothesis applied to e' , the inductive hypothesis $F^{i-1}(\perp) \lesssim \mathcal{D}[[e]]\eta_d$, and the assumption $c_1 \lesssim d_1$), we obtain, by Lemma 6.2,

$$\mathcal{C}[[e']] \eta_c[\text{self} \mapsto F^{i-1}(\perp), x \mapsto c_1] \ c_2 \dots c_n \sqsubseteq \mathcal{D}[[e']] \eta_d[\text{self} \mapsto \mathcal{D}[[e]]\eta_d, x \mapsto d_1] \bullet d_2 \bullet \dots \bullet d_n$$

Thus,

$$F^i(\perp) \ c_1 \dots c_n \sqsubseteq \mathcal{D}[[e]]\eta_d \bullet d_1 \bullet \dots \bullet d_n$$

and, by Lemma 6.2, $F^i(\perp) \lesssim \mathcal{D}[[e]]\eta_d$.

■

6.3 Discussion

From the two theorems, it follows that, for all closed basic-valued expressions e , $\mathcal{C}[[e]]\eta_\perp = \mathcal{D}[[e]]\eta_\perp$. While this result guarantees that the two semantics give the same computed results for all programs, it falls short of what is ideally desired: a direct correspondence between the higher-type abstractions (for objects) constructed by the two semantics.

Intuitively, it seems that the correspondence should be precisely what is stated in Theorem 6.1, condition 3. A \mathcal{D} -object d and a \mathcal{C} -object c are equivalent if $d \ d$ and c are equivalent

as functions. However, strengthening conditions 1 and 2 to represent equivalence poses harder theoretical problems. The “downward closed” property used in the establishing the existence of the relation becomes unavailable and it seems that the approximations of d in iterative construction of D_∞ do not grow as fast as the corresponding approximations of c in the constructions of C_∞ . Some other technique must be found for establishing the existence of the equivalence relation. Secondly, the proof of Theorem 6.1 also uses the fact that the relation is an approximation rather than an equivalence. To strengthen it, one would have to find approximate forms of objects which grow in parallel in the two semantics. (Our obj^i expressions denote approximate objects only in the \mathcal{D} -semantics, not in the \mathcal{C} -semantics). We leave these tantalizing problems open.

7 Conclusion

We presented two semantic models of object-oriented languages focusing on the inheritance aspects. The closure model completely hides instance variables and gives an abstract semantic framework. The data structure model treats objects as structures containing both data and operations, and provides a somewhat lower level semantic framework.

At a deeper level, the closure model treats `self` via explicit fixed point operators while the data structure models uses self-application familiar from the untyped lambda calculus. Thus, the two models may also be appropriately called the fixed point and self-application models. The relationship between these two approaches brings to surface the sophisticated semantic structure inherent in object-oriented programming languages. Because self-application uses universal reflexive domains, the fixed point approach is better suited for typed languages.

Inheritance in the sense of SMALLTALK-80 has a somewhat awkward presentation in the fixed point approach. A class has two levels of meaning: as seen by its subclasses and as seen by the instances. On the other hand, the self-application model treats inheritance in a more straightforward fashion, but its apparent simplicity may be deceptive.

Our study leads to important theoretical questions regarding the relationship between self-application and fixed point methods of semantic analysis, some of which are still unanswered. Further work is needed to clarify this relationship.

Appendix

We discuss the technical details of the construction of the reflexive domains involved in Section 6 and the existence proof of the relations involved in Theorems 6.1 and 6.2. The essential technique of constructing reflexive domains using embedding-projection pairs is due to Scott [Sco72] and a more expository treatment is found in [MP87, Rey72, SP82, Wad76]. In the following, we use the notation of [MP87] uniformly.

Construction of D We would like to define a domain satisfying the isomorphism

$$D \cong B + [D \rightarrow D \rightarrow D]$$

where B is some arbitrary domain and $+$ is the separated sum construction. For this purpose, we first define an ω -chain of domains D_0, D_1, \dots and a pair of maps between consecutive domains of the form $f_n : D_n \rightleftarrows D_{n+1} : f_n^R$. (f_n is an *embedding* and f_n^R is a *projection* satisfying $f_n^R \circ f_n = \text{id}$ and $f_n \circ f_n^R \sqsubseteq \text{id}$). The definitions are as follows:

$$\begin{aligned} D_0 &= \{\perp\} \\ D_{n+1} &= B + [D_n \rightarrow D_n \rightarrow D_n] \\ f_0 &= \lambda d. \perp & f_0^R &= \lambda d. \perp \\ f_{n+1} &= \text{id}_B + (\lambda d. \lambda a. f_n \circ d(f_n^R a) \circ f_n^R) & f_{n+1}^R &= \text{id}_B + (\lambda d. \lambda a. f_n^R \circ d(f_n a) \circ f_n) \end{aligned}$$

where $f + g$ is the notation for the function that maps \perp to \perp and the two summands of the domain to the respective summands of the codomain by f and g respectively. We can define embedding-projection pairs between arbitrary domains in the chain by $f_{nm} = f_{m-1} \circ \dots \circ f_n$ and $f_{nm}^R = f_n^R \circ \dots \circ f_{m-1}^R$.

The domain D is the colimit of the chain D_i . An element of D is an infinite sequence $d_0 d_1 d_2 \dots$ of values drawn from the domains D_i such that, for every n , $d_n = f_n^R(d_{n+1})$. There is a cone of embedding-projection pairs from the chain to D defined by:

$$\begin{aligned} \mu_n : D_n &\rightleftarrows D : \mu_n^R \\ (\mu_n(d_n))_m &= \begin{cases} f_{nm}(d_n) & (m \geq n), \\ f_{mn}^R(d_n) & (m < n) \end{cases} \\ \mu_n^R(d) &= d_n \end{aligned}$$

Note that μ_n is the inclusion of D_n in D . Further, we can define a projection $P_n : D \rightarrow D$ which projects D to the image of D_n in D . This is defined by $P_i = \mu_n \circ \mu_n^R$.

By applying the functor $T(D) = B + [D \rightarrow D \rightarrow D]$ to the above cone, we obtain another cone from the subchain D_1, D_2, \dots to $B + [D \rightarrow D \rightarrow D]$. The embedding-projection pairs of the latter cone are defined by:

$$\begin{aligned} \nu_n : D_{n+1} &\rightleftarrows B + [D \rightarrow D \rightarrow D] : \nu_n^R \\ \nu_n &= \text{id}_B + (\lambda d. \lambda a. \mu_n \circ d(\mu_n^R a) \circ \mu_n^R) & \nu_n^R &= \text{id}_B + (\lambda d. \lambda a. \mu_n^R \circ d(\mu_n a) \circ \mu_n) \end{aligned}$$

Then the isomorphism pair $\Phi : D \rightleftarrows B + [D \rightarrow D \rightarrow D] : \Psi$ is given by the formulae:

$$\Phi = \bigsqcup_{n \geq 0} \nu_n \circ \mu_{n+1}^R \quad \Psi = \bigsqcup_{n \geq 0} \mu_{n+1} \circ \nu_n^R$$

It is easy to calculate that

$$(d \bullet (\mu_n d'))_n = (\Phi(d) d d')_n = d_{n+1} d_n d'$$

We often write $\Phi(d)$ is simply as d .

Construction of C The construction of the domain C satisfying

$$C \cong B + [C \rightarrow C]$$

is similar to that of D . The essential difference is in the embedding-projection pairs:

$$\begin{aligned} f_n : C_n &\xleftrightarrow{\quad} C_{n+1} : f_n^R \\ f_0 &= \lambda d. \perp & f_0^R &= \lambda d. \perp \\ f_{n+1} &= \text{id}_B + (\lambda d. f_n \circ d \circ f_n^R) & f_{n+1}^R &= \text{id}_B + (\lambda d. f_n^R \circ d \circ f_n^R) \end{aligned}$$

Construction of the logical relations The technique used here is that of [MP87][Appendix II]. First, define relations $\lesssim^n \subseteq D_n \times C$ as follows: $d \lesssim^n c$ if and only if one of the following holds:

1. $d = \perp$,
2. $d \in B, c \in B$ and $d \sqsubseteq c$, or
3. $n > 0, d \in [D_{n-1} \rightarrow D_{n-1} \rightarrow D_{n-1}], c \in [C \rightarrow C]$ and

$$\forall d' \in D_{n-1}, c' \in C. d' \lesssim^{n-1} c' \Rightarrow d \bullet_n d' \lesssim^{n-1} c c'$$

where $d \bullet_n d' = d(f_{n-1}^R d) d'$.

Some elementary properties of the \lesssim^n relations follow:

Lemma 8 Each \lesssim^n is downward closed in the first argument, *i.e.*, $d \lesssim^n c$ and $d' \sqsubseteq d$ implies $d' \lesssim^n c$.

Proof: Easy induction on n .

Lemma 9 Each \lesssim^n is inclusive in the first argument, *i.e.*, if d_i is an increasing sequence in D_n and $d_i \lesssim^n c$ then $(\bigsqcup_i d_i) \lesssim^n c$.

Proof: By induction on n .

Lemma 10 The relations \lesssim^n respect the embedding-projection pairs between D_n 's. That is,

1. $d \lesssim^n c \Rightarrow f_n(d) \lesssim^{n+1} c$.
2. $d \lesssim^{n+1} c \Rightarrow f_n^R(d) \lesssim^n c$.

Proof: By simultaneous induction on n . For (1), $d \lesssim^n c$ gives three cases: If $d = \perp$ then $f_n(d) = \perp \lesssim^{n+1} c$. If $d \in B$ then $f_n(d) = d$ and, since $d \sqsubseteq c$, $d \lesssim^{n+1} c$. The final case is that of $d \in [D_{n-1} \rightarrow D_{n-1} \rightarrow D_{n-1}], c \in [C \rightarrow C]$ and $\forall d'' \in D_{n-1}, c'' \in C. d'' \lesssim^{n-1} c'' \Rightarrow d \bullet_n d'' \lesssim^{n-1} c c''$. To show the result, assume $d' \in D_n$ and $c' \in C$ such that $d' \lesssim^n c'$. First, calculate

$$\begin{aligned} (f_n d) \bullet_{n+1} d' &= (f_n d)(f_n^R(f_n d))d' \\ &= (f_n d) d d' \\ &= (f_{n-1} \circ d(f_{n-1}^R d) \circ f_{n-1}^R) d' \\ &= f_{n-1}(d \bullet_n (f_{n-1}^R d')) \end{aligned}$$

By inductive hypothesis (2), $f_{n-1}^R d' \lesssim^{n-1} c'$. So, by assumption, $(d \bullet_n (f_{n-1}^R d')) \lesssim^{n-1} cc'$. By using the inductive hypothesis (1), we obtain $f_{n-1}(d \bullet_n (f_{n-1}^R d')) \lesssim^n cc'$. Thus, for all $d' \in D_n$, $c' \in C$, $d' \lesssim^n c' \Rightarrow (f_n d) \bullet_{n+1} d' \lesssim^n cc'$.

For (2), $d \lesssim^{n+1} c$ gives three cases: If $d = \perp$ or $d \in B$, the result is obvious. For the case where $d \in [D_n \rightarrow D_n \rightarrow D_n]$, $c \in [C \rightarrow C]$ and $\forall d'' \in D_n, c'' \in C. d'' \lesssim^n c'' \Rightarrow d \bullet_{n+1} d'' \lesssim^n cc''$. As usual, assume $d' \in D_{n-1}$, $c' \in C$ such that $d' \lesssim^{n-1} c'$. Calculate

$$\begin{aligned} (f_n^R d) \bullet_n d' &= (f_n^R d)(f_{n-1}^R (f_n^R d))d' \\ &= (f_{n-1}^R \circ (d(f_{n-1}(f_{n-1}^R (f_n^R d)))) \circ f_{n-1}) d' \\ &\sqsubseteq (f_{n-1}^R \circ (d(f_n^R d)) \circ f_{n-1}) d' \\ &= f_{n-1}^R (d \bullet_n (f_{n-1} d')) \end{aligned}$$

By inductive hypothesis (1), $f_{n-1} d' \lesssim^n c'$. So, by assumption, $d \bullet_n (f_{n-1} d') \lesssim^n cc'$. By inductive hypothesis (2), $f_{n-1}^R (d \bullet_n (f_{n-1} d')) \lesssim^{n-1} cc'$. Since \lesssim^{n-1} is downward closed, we obtain the result $(f_n^R d) \bullet_{n-1} d' \lesssim^{n-1} cc'$. \square

Proof of Theorem 6.1 We define the relation $\lesssim \subseteq D \times C$ by

$$d \lesssim c \text{ iff } \forall n. d_n \lesssim^n c$$

This is clearly inclusive using Lemma 7. Next, we show that it satisfies the recursive specification given in Theorem 6.1, *i.e.*, $d \lesssim c$ iff

1. $d = \perp$,
2. $d \in B, c \in B$, and $d \sqsubseteq c$, or
3. $d \in [D \rightarrow D \rightarrow D]$, $c \in [C \rightarrow C]$, and, for all d' and c' , $d' \lesssim c' \Rightarrow d \bullet d' \lesssim cc'$.

(d in the above conditions is really Φd , but we ignore to write this coercion).

\Rightarrow Suppose $d \lesssim c$. There are three possibilities for d . If $d = \perp$, there is nothing to be proved. If $d \in B$, $\mu_n^R(d) = d$ for all $n > 0$. By assumption $d \lesssim^n c$ and this shows that $c \in B$ and $d \sqsubseteq c$. If $d \in [D \rightarrow D \rightarrow D]$, $d_{n+1} \in [D_n \rightarrow D_n \rightarrow D_n]$ for all n . Since $d_{n+1} \lesssim^{n+1} c$, we have that $c \in [C \rightarrow C]$. Assume $d' \in D, c' \in C$ such that $d' \lesssim c'$. By definition, $d'_n \lesssim^n c'$ and, so, $d_{n+1} \bullet_n d'_n \lesssim^n cc'$. Since $(d \bullet d')_n = d_{n+1} \bullet_n d'_n$, by definition, $d \bullet d' \lesssim cc'$.

\Leftarrow If $d = \perp$ then $\mu_n^R(d) = \perp$. Since $\perp \lesssim^n c$, we have $d \lesssim c$. If $d \in B, c \in B$ and $d \sqsubseteq c$ then $\mu_0^R(d) = \perp$ and $\mu_{n+1}^R(d) = d$. So, we have $\mu_n^R(d) \lesssim^n c$ for all n . Finally, suppose $d \in [D \rightarrow D \rightarrow D]$, $c \in [C \rightarrow C]$ and $\forall d'' \in D, c'' \in C. d'' \lesssim c'' \Rightarrow d \bullet d'' \lesssim cc''$. We need to show that $d_n \lesssim^n c$ for all n . For $n = 0$ the result is trivial. For $n > 0$, Consider $d' \in D_{n-1}$ and $c' \in C$ such that $d' \lesssim^{n-1} c'$. We have $\mu_{n-1}(d') \lesssim c'$ and, by assumption, $d \bullet \mu_{n-1}(d') \lesssim cc'$. The definition of \lesssim gives $(d \bullet \mu_{n-1}(d'))_{n-1} \lesssim^{n-1} cc'$, but, the LHS is the same as $d_n \bullet_{n-1} d'$. Thus, for all $d' \in D_{n-1}, c' \in C, d' \lesssim^{n-1} c' \Rightarrow d_n \bullet_{n-1} d' \lesssim cc'$. This shows $d_n \lesssim^n c$ for all n . \square

The construction of the logical relation mentioned in Theorem 6.2 proceeds similarly.

References

- [ASS85] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Mass., 1985.

- [Car84] L. Cardelli. A semantics of multiple inheritance. In *Semantics of Data Types*, pages 51–67. Springer-Verlag LNCS Vol. 173, 1984.
- [Coo89] W. R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Dep. of Computer Science, Brown Univ., May 1989. (Tech. Report CS-89-33).
- [CP93] W. Cook and Palsberg. A denotational semantics of inheritance and its correctness. *Information and Computation*, 1993. (to appear).
- [DN66] O.-J. Dahl and K. Nygaard. An Algol-based simulation language. *Communications of the ACM*, 9(9):671–678, Sep 1966.
- [DoD82] DoD. *Reference Manual for the ADA Programming Language*. United States Department of Defense, 1982.
- [GR83] A. Goldberg and D. Robson. *SMALLTALK-80: The Language and its Implementation*. Addison-Wesley, Reading, MA, 1983.
- [Gun92] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. MIT Press, 1992.
- [Kam88] S. Kamin. Inheritance in SMALLTALK-80: A denotational definition. In *ACM Symp. on Princ. of Program. Lang.*, January 1988.
- [Kam90] S. N. Kamin. *Programming Languages: An Interpreter-Based Approach*. Addison-Wesley, Reading, MA, 1990.
- [KL85] R. M. Keller and G. Lindstrom. Approaching distributed database implementations through functional programming concepts. In *Intl. Conf. on Distributed Computing Systems*. IEEE, May 1985.
- [LAB⁺81] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. C. Schaffert, R. Scheffler, and A. Snyder. *CLU Reference Manual*, volume 114 of *Lect. Notes in Comp. Science*. Springer-Verlag, 1981.
- [Mil74] R. E. Milne. *The Formal Semantics of Computer Languages and their Implementation*. PhD thesis, Univ. of Cambridge, 1974.
- [MP87] P. D. Mosses and G. D. Plotkin. On proving limiting completeness. *SIAM J. Computing*, 16(1):179–194, Feb 1987.
- [MS76] R. E. Milne and C. Strachey. *A Theory of Programming Language Semantics*. Chapman and Hall, London, 1976.
- [Par70] D. M. R. Park. The \mathbf{y} -combinator in Scott’s lambda-calculus models. In *Symposium on Theory of Programming*, Warwick, 1970. University of Warwick. (unpublished).
- [Red88] U. S. Reddy. Objects as closures: Abstract semantics of object oriented languages. In *ACM Symp. on LISP and Functional Programming*, pages 289–297. ACM, July 1988.

- [Rey72] J. C. Reynolds. Notes on a lattice-theoretic approach to the theory of computation. Lecture notes, Systems and Information Science, Syracuse University, Oct 1972.
- [Rey74] J. C. Reynolds. On the relation between direct and continuation semantics. In *Intern. Colloq. Automata, Languages. and Programming*, pages 141–156. Springer-Verlag, Berlin, 1974.
- [Sco72] D. S. Scott. Continuous lattices. In F. W. Lawvere, editor, *Toposes, Algebraic Geometry and Logic*, pages 97–136. Springer-Verlag, 1972. (Lecture Notes in Mathematics, Vol. 274).
- [SP82] M. B. Smyth and G. D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM J. Computing*, 11:761–783, 1982.
- [Sto77] J. E. Stoy. *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory*. MIT Press, 1977.
- [Str86] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 1986.
- [Wad76] C. P. Wadsworth. The relation between computational and denotational properties for Scott’s D_∞ -models of the lambda calculus. *SIAM J. Computing*, 5:48–521, 1976.
- [Weg87] P. Wegner. Dimensions of object-based language design. In *Object Oriented Prog. Systems, Lang. and Applications*. ACM SIGPLAN, 1987.