Semantics of Object-Oriented Languages[1]

The rise in popularity of the object-oriented paradigm is relatively recent. Formal semantics for this paradigm is therefore not nearly are well developed as it is for the more traditional paradigms we have examined so far. The variety of semantic approaches we have developed for traditional languages is absent in this case. In this presentation, the denotational approach is pursued for object-oriented languages through a series of successively expanding steps.

ObjectTalk

The initial step provides the analysis for a very simple language that Kamin and Reddy call *ObjectTalk*. This language does not include "classes", although methods can create objects each time they are called, so some similar effects can be achieved. The only constructs in this language are defining objects, and sending them messages.

The semantic domains are as follows:

- □ □ loc
- □ □ env = variable □ loc
- □ □ state = loc □ val
- e □ expression = env □ state □ (val □ state)
- □ □ method = state □ val* □ (val □ state)
- □ □ menv = message □ method
- o □ objectval = menv
- □ □ classval = state □ (menv □ state)
- □ □ superclassval = state □ (env □ (menv □ menv) □ state)
- v □ val = basicval + loc + objectval + classval + superclassval

It is assumed that 'basicval' includes usual pre-defined values (e.g., integers) for which definitions are omitted. The domain 'loc' can be thought of as pointers or machine addresses and will not be elaborated. The domains 'expression' and 'message' are syntactic, and any conventional message identifier syntax is acceptable. The domains will be discussed further as we proceed.

The syntax for the definition of objects is the **object expression**, written as

e ::= **obj**(x_1, … , x_n) {$m_1(\overline{y}_1)=e_1$, … , $m_k(\overline{y}_k)=e_k$}

where x_1, … , x_n are local (i.e., instance) variables of the object, m_1, … , m_k identify the messages, and provide the method definitions. The $\overline{y}_1$, … , $\overline{y}_k$ are parameter lists (the overbar is used to denote comma separated lists) for the methods, and a method body can refer to both its parameters and the instance variables.

[1] This presentation is based on "Two semantic models of object-oriented languages" by S. N. Kamin & U. S. Reddy that appears in *Theoretical Aspects of Object-Oriented Programming*, C. A. Gunter & J. C. Mitchell, Eds.

Each occurrence of an object expression signals the creation of a corresponding object. With these conventions objects are denoted only by object expressions and are therefore anonymous.

The second (and final) syntax category of ObjectTalk is the **message expression**, written as

$$e ::= e_0.m(\bar{e}_a)$$

where e_0 is the *receiver object*, m is the message, and $\bar{e}_a$ is a list of argument expressions.

There is also a reserved word, **self**, that may appear in method bodies and is understood to denote the object receiving the message of the method currently being executed. While it is technically unnecessary in the present context where a single object is endowed with methods, **self** is introduced here to smooth the transition to the next semantic step where *classes* endow a collection of objects with methods defined only once.

Example.
The following illustrates the definition of a (2-dimensional) point object.

```
p = obj(x,y) {put(a,b) = begin x:= a; y:= b end,                    (6)
              dist(0 = sqrt(sqr(valof x) + sqr(valof y)),
              closer(q) = self.dist() < q.dist()}
```

The object p declares two instance variables, x and y to represent the coordinates of the point. It also defines three messages. The 'put' message has two parameters and sets the coordinates of a point (creation leaves them undefined); the 'dist' message has no parameters and gives the distance of the point from the origin; and 'closer' has one “point-like” parameter q, and tests if the receiving point is closer to the origin than q.

For the sake of brevity, we are taking the liberty of attaching an identifier to the indicated point object. This naming is “outside” the convention of the semantics being developed. In particular, the identifier 'p' is not permitted within the body of the point definition.

Several traditional semantic issues are being disregarded in order to focus on the issue: “what should objects denote?”. The point of view that is taken here is an external behavioral view — an object responds to sequences of messages. Therefore, the meaning of an object can simply be an environment binding messages to their methods. Hence,

objectval = menv = message → method, where
method = state × val* → (val × state)

This is a significant choice in the semantic definition. The state of an object is not providing a direct contribution to its meaning — the object meaning only maps messages to methods. This recognizes the encapsulation intention of the object -oriented paradigm, where the internal storage of an object is only directly accessible within the object, and the only effect in client code is experienced through message responses. However, this complicates the semantic description of objects since the correspondence

of message to method is not constant — a message at one point may exhibit different behavior than the same message at another point where the internal state has changed. The method domain is similar to what has previously been used for procedures in denotational semantics. In particular, the environment at the point of message sending (i.e., method invocation) does not effect the meaning. These characteristics must be reflected in the message to method correspondence.

So the first attempt to express the meaning of an object expression in terms of its action for a message, is in the context of a state that is the first argument to the resulting method. The approach here is that the meaning of an object expression extends the current environment ρ to allocate space for the instance variables, bind the variables to the newly allocated space, producing environment ρ_1, then produce the final environment ρ_2 where the message identifiers are bound to the method bodies preformed in environment ρ_1. Symbolically,

meaning $[\![\textbf{obj}(x_1, \ldots, x_n)\ \{m_1(\bar{y}_1), \ldots, m_k(\bar{y}_k)\}]\!]\ \rho\ \sigma =$

extendEnv(ρ_1, $\bar{m}$, $\overline{meth}$)

where ρ_1= entendEnv(ρ, $\bar{x}$, $\overline{loc}$) , allocate $\overline{\sigma'}$ = <σ',$\overline{loc}$>, and
$meth_i$ = perform $[\![body_i]\!]\ \rho_1$.

This semantics is not fully satisfactory since we would like to be able to refer to an object's messages in the methods defining those messages. This is not possible in the above definition because the methods are executed in the environment ρ_1 while the message environment binding the message names to methods is the result ρ_2 =

extendEnv(ρ_1, $\bar{m}$, $\overline{meth}$).

In order to accomplish the recursive referral, the reserved word **self** is bound to the message environment being constructed for the object. That is, **self** designates the message to method mapping being constructed, and its definition is therefore inherently recursive. This can be expressed either by explicit recursion, or by applying a "fixed point operator". Kamin and Reddy use the fixed point operator (recursively, extendEnv(ρ_2, **self**, ρ_2) = ρ_2).

meaning $[\![\textbf{obj}(x_1, \ldots, x_n)\ \{m_1(\bar{y}_1), \ldots, m_k(\bar{y}_k)\}]\!]\ \rho\ \sigma =$
fix_{ρ} (extendEnv(ρ_2, **self**, ρ)) (7)

where ρ_1= entendEnv(ρ, $\bar{x}$, $\overline{loc}$) , allocate $\overline{\sigma'}$ = <σ',$\overline{loc}$>, and
$meth_i$ = perform $[\![body_i]\!]$ entendEnv(ρ_1, **self**, fix_{ρ}), and

ρ_2 = extendEnv(ρ_1, $\bar{m}$, $\overline{meth}$).

Finally, the meaning of a message send reflects evaluating the receiver object to obtain an appropriate state, and message mapping function, evaluating the argument expressions in the resulting state, and then invoking the method designated by the mapped message with those arguments.

meaning $[\![e_0.m(\overline{e}_a)]\!]\ \square\ \square = \square\ m\ \overline{v}\ \square_2$ (8)

where meaning $[\![\ e_0]\!]\ \square\ \square = <\square,\square_1>$, and meaning $[\![\overline{e}_a]\!]\ \square\ \square_1 = <\overline{v},\square_2>$.

ClassTalk

In this language, Kamin and Reddy introduce classes, but not inheritance. The new syntax providing **class expressions** is

e ::= **class**$(x_1, \ldots, x_n)\ \{m_1(\overline{y}_1)=e_1, \ldots, m_k(\overline{y}_k)=e_k\}$

where $x_1, \ldots, x_n$ are local (i.e., instance) variables of the object, $m_1, \ldots, m_k$ identify the messages, and provide the method definitions. The $\overline{y}_1, \ldots, \overline{y}_k$ are parameter lists for the methods, and a method body can refer to both its parameters and the instance variables.

They also add the means to create instance objects of classes, namely

e ::= **new** e_c

where e_c is a class expression.

The semantics Kamin and Reddy attach tor this new construct is is still that of a message environment (mapping messages to methods), namely

meaning $[\![$ **new class**$(x_1, \ldots, x_n)\ \{m_1(\overline{y}_1)=e_1, \ldots, m_k(\overline{y}_k)=e_k\}]\!] =$

meaning $[\![$ **obj**$(x_1, \ldots, x_n)\ \{m_1(\overline{y}_1)=e_1, \ldots, m_k(\overline{y}_k)=e_k\}]\!]$ (11)

In the light of (11), classes do not add any expressive capacity, but they are still valuable from a practical perspective. But this still does not provide for importation of another class, a capability we would certainly desire even in the absence of inheritance.

For the treatment of inheritance, see the paper cited that goes on to describe a third language, InheritTalk, that incorporates this capability.