ANC 18/01/92

The Semantics Of Smalltalk

The semantics of Smalltalk is developed in terms of an algebra and a machine which executes with respect to the algebra. The carriers of the algebra contain the data elements with which Smalltalk computes, and the operations define how the elements are born, transformed and combined. The machine extends the algebra with an evaluation mechanism. The state of the machine represents a snapshot of Smalltalk evaluation.

We are interested in finding the SECD builtins and extensions to the evaluation mechanism which allows Smalltalk programs to be viewed as sugared λ-calculus. These are found by defining a homomorphism which maps a Smalltalk semantics, in terms of the algebra and machine, to an $SECD^+$ semantics.

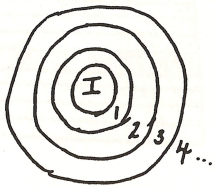
Whilst the $SECD^+$ builtins are peculiar to Smalltalk the evaluation extensions are proposed as a characteristic feature of OOPL's. The SECD machine is extended with environment reification in order to execute Smalltalk programs.

Overview of Smalltalk

A Smalltalk object is a collection of methods, which are parameterized expressions rather like functions. Each method is associated with an environment which contains bindings for the identifiers referenced in the method body. A Smalltalk class defines how to construct an object, which is called an instance of the class. The class dictates the methods which comprise the object and how the method environments are composed. A method environment is constructed from several environments, some of which are freshly created and some of which are shared between different parts of the program.

A method is executed by sending the object which owns it a message. The body of the method is evaluated with respect to the associated environment. If an evaluating method were cut in half it would look like an onion. At the centre is the expression being evaluated, radiating out from the centre are concentric circles like the layers of onion skins. Each skin represents an environment. All the skins combine to form the method's environment.

Consider evaluating an identifie reference within the body of a method.

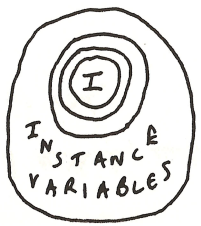


The search for the value of I starts in the environment marked 1. If this fails to find a binding then the search escapes through the skin to the next layer marked 2. Then 3 and 4 and so on. Each of the layers corresponds to the evaluation of different types of program fragments. The program fragments introduce identifie bindings which have varying scopes leading to layers or part-layers being shared between methods and objects.

Each type of layer is described in turn:

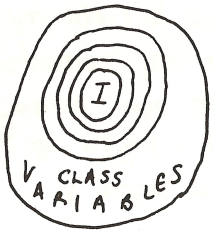
The innermost binding layer contains the locally defined identifiers of the method. These bindings are freshly generated each time the method is invoked and are forgotten when the method returns the value. The bindings are not shared.

This layer binds the formal parameters of the method. Like the locals these bindings are freshly generated each time the method is invoked, and are not shared.



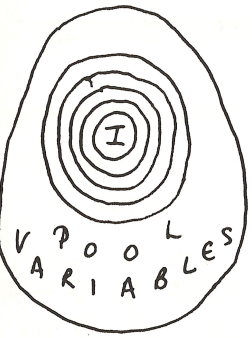
This layer contains bindings for the instance variables. The instance variable bindings are freshly generated when an instance of a class is created. The bindings are shared between all of the

methods of the object.



This layer contains bindings for class variables. Class variable bindings are created once when a class is created and shared between all instances of that class.

A class is created by extending an existing class, which is called the superclass. The extension includes class variable bindings, which are combined with those of the superclass. The resulting class variable bindings share with those of the superclass. The class variable bindings of instances of the new class will share with class variable bindings of instances of the superclass.



This layer contains bindings for pool variables. A Smalltalk pool is an environment; the identifiers bound in a pool are called pool variables. A class declares which pools are combined to form this layer in its instances. The bindings are freshly generated when the pool is created and all classes which reference the pool will share the bindings. When a new class is generated the extension will add to and share with the pools of the superclass.

This layer contains bindings for global variables. Global variables are shared everywhere.

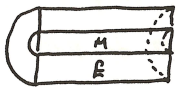
A method, without any associated environment, will be represented as a 3D rod,

An object is constructed by fitting methods together. Methods fit together by plugging the rods end-to-end,

The environment which an object associates with its methods starts with instance variables and radiates outwards in the onion diagrams. These environments are shared between all of the methods. An environment is a tube which is slid over a method rod. End on, a method and environment look like

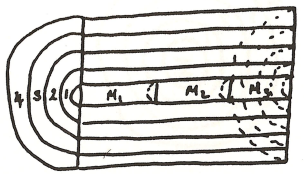
A licorice allsort!

A cut away section is,



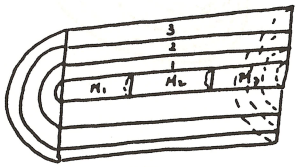
A method surrounded by its environment.

An object will look like



1 = instance variables
2 = class variables
3 = pool variables
4 = global variables

When a class is defined it describes the methods, instance variable names, class variables and pool variables which will be composed to make each of its instances. Suppose that C is an existing class whose instances look like,



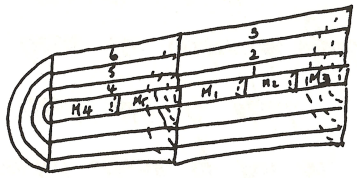
1 = Instance variables local to each instance.
2 = Class variables shared by each instance
3 = Pool variables shared by each instance (and any other class which uses the pool).

A new class C' is constructed by adding an extension to C. The extension is a collection of methods, some instance variable names, some class variables and some pools. An instance of C' will have the methods of C and the methods defined by the extension.

The methods of an instance of C', presuming the extension defines m_4 and m_5.

The instance variables, class variables and pool variables as defined by C will be scoped, as before, over the methods m_1, m_2 and m_3 of the instance of C'. The instance variables, class variables and pool variables of the extension will be scoped over the methods m_4 and m_5 but <u>not</u> over m_1, m_2 or m_3. Finally the instance variables, class variables and pool variables of C are scoped over the methods m_4 and m_5 by sliding the tubes down over the corresponding environments contributed by the extension.

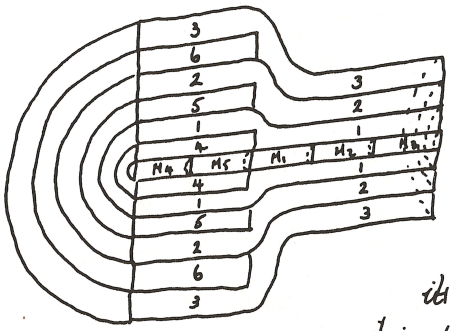
This can be done in two phases, first fill in the respective tubes,



Contributions from the extension.

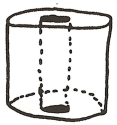
Contributions from C.

and then slide tubes 1, 2 and 3 down over their corresponding extensions 4, 5 and 6.



The completed instance of C' has a fat end and a thin end. When an object is combined in this way by having its environments "rolled down" then it is always the fat end which is rolled, ie 3+6 will be the pool environment and 2+5 will be the class environment etc.

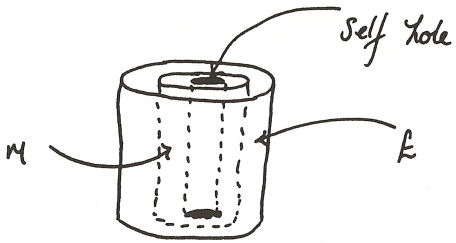
Each method of an object is associated with an object which it calls self. The methods are drilled down the centre to produce a hole.



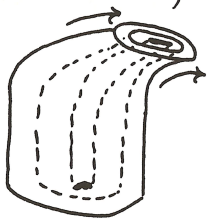
A method rod which has had a self hole drilled through it.

When methods are fitted together the holes join up to form one continuous hole,

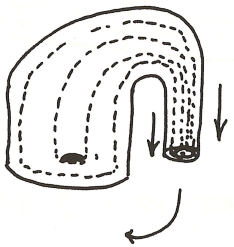
Environments are wrapped around the methods as usual,



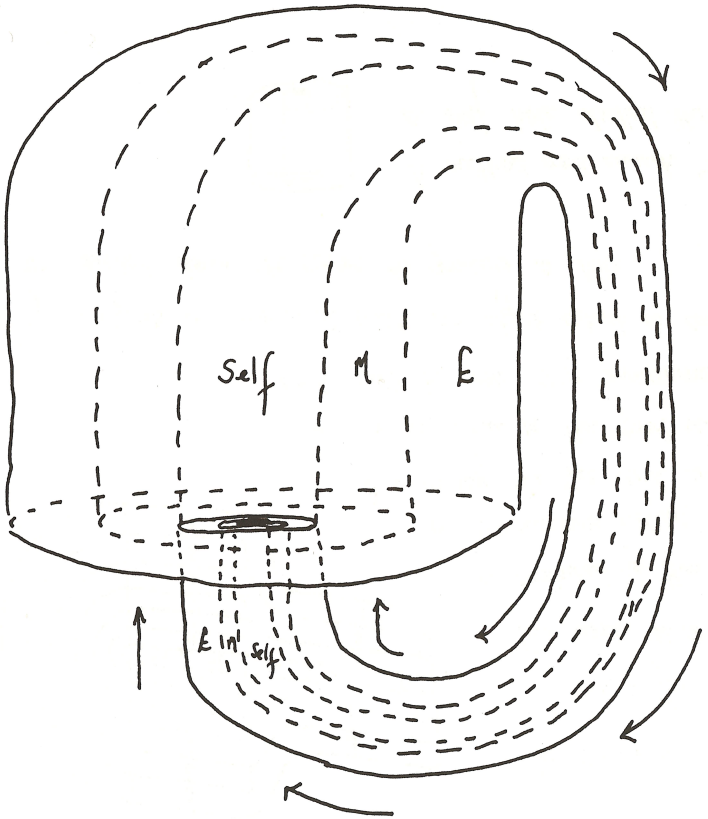
The self hole expects an object to be fed into it, and then the object is complete. Feed the object into its own self hole!



Start to pull the object round.



pull down and round again



The object is fed into itself. It is pushed as far as it will go. This will cycle endlessly, spiralling inwards. At all levels the object is self referential. Suppose a method, m, sends a message m to its self object. Then it will end up back where it started.

Suppose a class C' is created by extending a superclass C. When an instance of C' is created think of this as,

(i) Create the C object

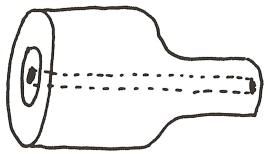
(ii) Create the extension object

(iii) Slot them together

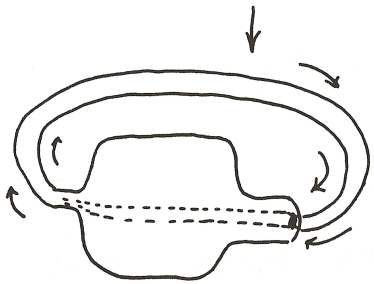
(iv) Feed the object into itself fat end first.

(ii) Create the extension object

(i) Create the C object



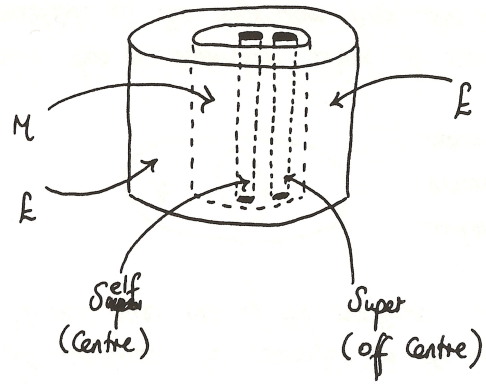
(iii) Slot them together



(iv) Feed the object into itself fat end first.

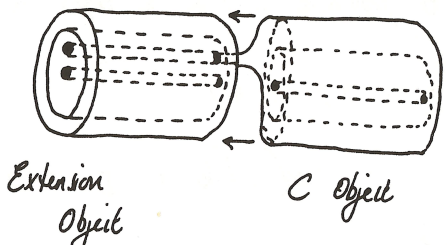
Each method is associated with an object which it calls Super. For a class C' which has been defined by adding an extension to a superclass C the Super object for the methods m_4 and m_5, defined by the extension, will be the part of the instance of C' contributed by C. The part of the instance of C' contributed by C, before Self is filled in, is shown as (i) above.

All methods defined in an extension have aligning holes drilled in them for Super.



Part of a drilled extension object.

When the extension object and the C object are slotted together, the C object is also fitted into the extension object's Super hole. This is difficult to draw but,



The C object is fed into the Super hole of the extension object. The C object is also slotted on to the extension object and all the environments are slid down. The Self holes of the extension object and the C object join up as before. At this point we have a C' object ~ an empty Self hole but a full Super hole.

The object is fed into itself fat end first as before to complete the instance of C'.

A Smalltalk Algebra

Smalltalk is described in terms of a simple algebra. A machine will be described which computes using the algebra. The machine will execute the Smalltalk evaluation mechanism.

Let Env_ be a function which maps types, α, to sets of environments which bind identifiers to values of type α. An environment is an ordered collection of bindings. The simplest environment is empty,

$$\underline{\text{nullenv}}\ \alpha : \text{Env}\ \alpha$$

A single binding is constructed using the function $(\mapsto \alpha)$

$$\_\ (\mapsto \alpha)\ \_ : I \to \alpha \to \text{Env}\ \alpha$$

For example

$$i\ (\mapsto \text{Class})\ \underline{\text{nullclass}} : \text{Env Class}$$

Environments are composed using the function $(\oplus \alpha)$

$$\_\ (\oplus \alpha)\ \_ : (\text{Env}\ \alpha) \times (\text{Env}\ \alpha) \to (\text{Env}\ \alpha)$$

where

$\underline{\text{nullenv}}\ \alpha$ is both the left and right identity of $(\oplus \alpha)$

The value associated with an identifier in an environment is extracted using the $(\cdot \alpha)$ function,

$$\_\ (\cdot \alpha)\ \_ : (\text{Env}\ \alpha) \times I \to \alpha$$

where

bindings shadow to the right.

Let Seq_ be a function which maps types, α, to a type containing sequences of values of type α. The simplest sequence is empty,

A sequence of length 1 is constructed using the function $(I\_I\ \alpha)$,

$$I\_I\ \alpha : \text{Seq}\ \alpha$$

Two sequences are concatenated using the function $(@\ \alpha)$

$$\_\ (@\ \alpha)\ \_ : (\text{Seq}\ \alpha) \times (\text{Seq}\ \alpha) \to (\text{Seq}\ \alpha)$$

An element is added to the head of a sequence using the function $(::\alpha)$

$$\_\ (::\alpha)\ \_ : \alpha \times (\text{Seq}\ \alpha) \to (\text{Seq}\ \alpha)$$

$[\ a, b, c, \ldots\ ]$ is sugar for,

$$a :: (b :: (c :: \ldots.))\quad \text{where } a, b, c : \alpha$$

Where the value of the type argument can be omitted, ie

$$I \mapsto 1\phi : \text{Env}$$

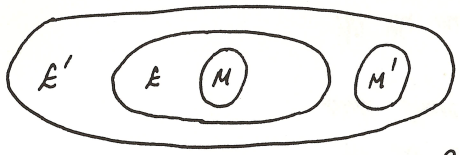
rather than

$$I\ (\mapsto \text{Int})\ 1\phi :$$

A class is (at least) a pair of environ
variable bindings, a collection of instance v
environment of methods. The identifiers in
variables are scoped over all of the method

A class also has a superclass whose E

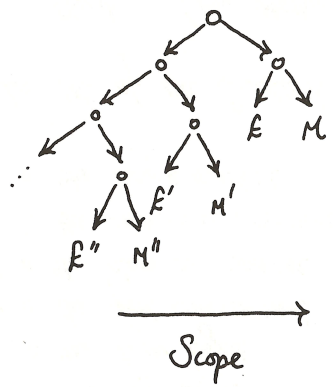
So the class and its superclass is,



The subclass is defined by adding an extension with E and M components to a superclass with E' and M' components.

The superclass has its own superclass whose E" component is scoped over M", M' and M.

A class is a tree structure,



The E components are scoped over all of the M components to their right in the tree.

Let Class be the set of all Smalltalk classes. The simplest class is empty and contains no environments, instance variables or methods,

$$\underline{nullclass} : Class$$

Non empty classes are constructed using the function $\underline{class}$

$$\underline{class} : Class \times Body \longrightarrow Class$$

For a class, $\underline{class}\,(C, B)$, the class C is the superclass, shown as the left-hand subtree in the digram above, and B contains the E and M components of the extension.

Let B_1 = extension with E and M
B_2 = extension with E' and M'
B_3 = extension with E" and M"

then the tree diagram is represented as,

$$\underline{class}\,(\underline{class}\,(\underline{class}\,(\ldots, B_3), B_2), B_1)$$

A method, defined in a class, is a parameterized expression. There are, as yet, no environments associated with the method. Such a method will be called a simple method. Let SimpleMethod be the set of all Smalltalk simple methods. A simple method is constructed using the function $\underline{meth}$,

$$\underline{meth} : I \times (Seq\ Instruction) \longrightarrow SimpleMethod$$

where

Seq Instruction is the compiled method body.

An environment of simple methods is a value of type SimpleMEnv

$$SimpleMEnv = (Env\ SimpleMethod)$$

Let Value be the set of all values which are denoted by Smalltalk expressions.

Let ValEnv be the set of value environments,

$$ValEnv = (Env\ Value)$$

Let Body be the set of all Smalltalk class extensions. A body will contain the pool environment, class environment, instance variable names and the method environment. A body is constructed using the function $\underline{body}$

$$\underline{body} : I^* \longrightarrow (ValEnv \times ValEnv \times SimpleMEnv) \longrightarrow Body$$

An object is a collection of methods. Each method has a pool, class and instance variable environment. A method also has a Self and Super object.

Let Method be the set of Smalltalk methods. A method is constructed using the function meth,

$$meth : Object \times Object \times ValEnv \times ValEnv \times ValEnv \times SimpleMethod \longrightarrow Method$$

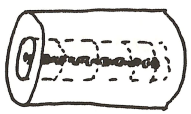
If $meth(o_1, o_2, pv, cv, iv, m)$ is a method then,

o_1 is the self object
o_2 is the Super object
pv is the pool environment
cv is the class environment
iv is the instance environment
m is the Simple method.

Let Object be the set of all Smalltalk objects. An object is an environment of methods,

$$Object = (Env \; Method)$$

Consider an object,



All the methods in the object will share the same Self object.

All the methods in an extension object will share the same environments.

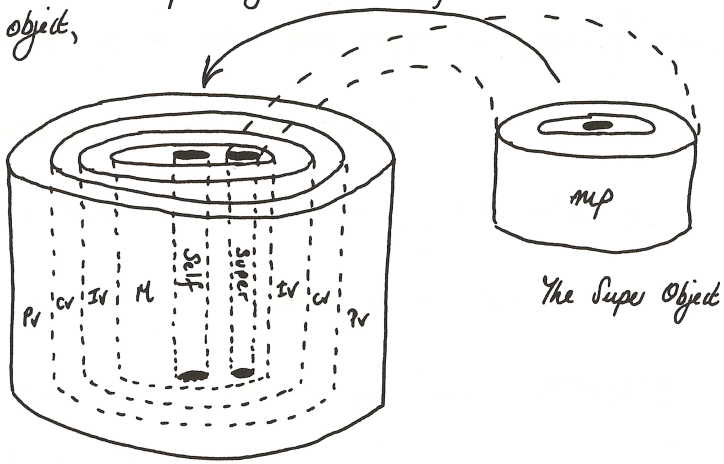
All the methods in the Super object of all the methods in the extension object will share the same Self object.

So the Self object is abstracted out as are the environments, producing a method package.

Let Mpkg be the set of all Smalltalk method packages. A method package is constructed using the function mpkg,

$$mpkg : Mpkg \times ValEnv \times ValEnv \times ValEnv \times SimpleMEnv \longrightarrow Mpkg$$

Let $mpkg(mp, pv, cv, iv, M)$ be a method package. This corresponds to an extension object whose Super object has been filled in with mp but which awaits a Self object,



The Extension Object

Method packages are added together using $\oplus$

$$- \oplus - : Mpkg \times Mpkg \longrightarrow Mpkg$$

This corresponds to placing two arbitrary objects, whose Self holes are vacant, together. No merging of environments occurs,