Semantics of *Smalltalk-80**

Mario Wolczko†
Dept. of Computer Science, The University, Manchester M13 9PL, United Kingdom
miw@uk.ac.man.cs.ux, mcvax!ukc!man.cs.ux!miw

Abstract

A formal model of the *Smalltalk-80* programming language is introduced. The semantics of much of the *Smalltalk-80* language are described using the denotational style. A poorly-designed feature of Smalltalk is highlighted, and alternative semantics are presented for the language feature.

1 Introduction

Smalltalk-80 is the archetypal object-oriented programming environment. All code within the *Smalltalk-80* system is written in the *Smalltalk-80* programming language (hereafter referred to as "Smalltalk"). Smalltalk has several characteristics that are common in object-oriented langauges, and others that distinguish Smalltalk from other languages: a uniformly-applied object model; an inheritance hierarchy of classes; and message-sending as the sole inter-object communication mechanism.

A formal model of Smalltalk would be useful in describing what Smalltalk is, and how it differs from other languages. Also, as many non-object-oriented languages have been described through formal models, it would help to illustrate how the object-oriented paradigm differs from other programming paradigms[5]. Furthermore, since a formal model is likely to be more concise and succinct than, say, an interpreter for the language[2], it would provide a convenient framework for discussion of language semantics. This is because a formal model would be at the right level of abstraction for such discussions: implementation details (e.g., the garbage collection strategy) would not appear in the formal model.

This paper introduces a formal model of Smalltalk. As Smalltalk is a non-trivial language, the complete model cannot be presented in the limited space available; the remainder will appear elsewhere[7]. However, the major parts of a simplified model are presented here. This simplified model also serves as an introduction to the more complicated model. Using the model as a basis, the design of Smalltalk is discussed, and weaknesses in the design are highlighted. In particular, the block evaluation mechanism is found to be inadequate, and an alternative design for this feature is presented.

2 A Formal Model of Smalltalk

The formal description technique used will be that of denotational semantics[1,3,6], "sugared" with a VDM-like syntax[4]. Several simplifications are made to keep the semantics tractable, and to preserve the right level of abstraction:

1. In a Smalltalk-80 system there a number of processes (conceptually) executing in parallel. Due to the lack of any protection mechanisms, these processes, which reside in a single address space, could interfere with each other. By convention, most methods in the Smalltalk-80 system assume that no interference is taking place, and the use of concurrency is limited to a few places in the system. To simplify the model we have assumed that no concurrency is present, and therefore that no interference can take place.

2. Non-local variables, i.e., global, class and pool variables, have been omitted. These could all be simulated in Smalltalk by sending messages to the appropriate dictionary.

*Smalltalk-80 is a registered trademark of Xerox Corp.

†Work supported by the Science and Engineering Research Council

3. The only forms of literal dealt with are integers and blocks. It would be straightforward to deal with other forms, such as Floats and Strings, but the treatment would be similar to that for integers, and is therefore omitted for brevity.

4. All the integer classes (SmallInteger, LargePositiveInteger, LargeNegativeInteger) have been replaced by a single class, Integer, instances of which are integers of arbitrary size.

5. A block in Smalltalk can be directed to return to the place that invoked its execution, or to the place that invoked its creation, which need not be the same. The semantics presented here only deal with the former case.

6. A block in Smalltalk can access all the variables of its enclosing method. This means that the activation record of a method that contains a block cannot be discarded when the corresponding method is exited. As a simplification, the semantics presented here restrict blocks so that they cannot access the temporaries of the enclosing method. However, they can access the arguments of the method and the instance variables of the receiver.

3 The Abstract Syntax of Smalltalk

A "program" in Smalltalk is composed of a set of class definitions. Each definition states: (i) how the class relates to other classes by inheritance, (ii) the instance variables that are defined by that class, and (iii) the messages that instances of that class respond to, and how. Formally, this can be stated thus:

$$Program = Class\_map$$

$$Class\_map = \text{map } Class\_name \text{ to } Class\_body$$

$$\begin{aligned} Class\_body :: \; & Instvars : \text{set of } Id \\ & Super : Superclass \\ & Methods : \text{map } Selector \text{ to } (Method\_body \cup Primitive\_method) \end{aligned}$$

In the single inheritance scheme supported by the Smalltalk Virtual Machine, each class can have zero or one superclasses:

$$Superclass = [Class\_name]$$

For the purposes of this paper, we shall restrict ourselves to the single inheritance scheme supported by the Smalltalk Virtual Machine.

A selector can be a single postfix identifier (a *unary* selector), an infix (*binary*) selector, or a *keyword* selector:

$$Selector = Unary \cup Binary \cup Keyword$$

$$Unary = Id$$

$$Binary = \{+, -, *, /, <, \ldots\}$$

$$Keyword = \text{seq of } Id$$

Later we will use a function, $nargs$: $Selector \rightarrow \mathbf{N}$, which returns 0 for unary selectors, 1 for binary selectors, and len s for a keyword selector s.

Each method body declares a number of arguments and temporary variables, and has a list of expressions as the executable part of the method.

$$\begin{aligned} Method\_body :: \; & Args : Ulist(Id)^1 \\ & Temps : \text{set of } Id \\ & Exprs : Expression\_list \end{aligned}$$

Expression_list = seq of *Expression*

An *Expression* can be one of four things: an assignment to a variable, an object name, a message send, or a literal object (integer or block):

Expression = *Assignment* ∪ *Object_name* ∪ *Message_list* ∪ *Literal_object*

Assignment :: *LHS* : *Id*
RHS : *Expression*

Object_name = *Id* ∪ {SELF, SUPER, ROOT}

SELF is the name of the receiver. SUPER is also an alias for the receiver, but messages sent to SUPER are searched for differently. ROOT is a global object guaranteed to be in the system. It plays the part of the global dictionary, Smalltalk.

A *Message_list* consists of an expression which evaluates to a receiver object, and a list of messages to be sent to the receiver (cascaded messages, to use the *Smalltalk-80* terminology).

Message_list :: *Rcvr* : *Expression*
Msgs : seq of *Message*

Message :: *Sel* : *Selector*
Args : *Expression_list*

A *Literal_object* is a 'constant' (i.e., immutable) object. Here, we only consider integers and blocks.

Literal_object = **Z** ∪ *Block_body*

One can consider a block to be a nameless method; it is activated by sending it the value message. Bound blocks are first-class objects. To emphasize the similarity between methods and blocks, we use the same abstract syntax for both:

Block_body :: *Args* : *Ulist*(*Id*)
Temps : set of *Id*[2]
Exprs : *Expression_list*

An extra field required in the full model, *RetHome*: **B**, has been omitted. A Smalltalk block differs from an anonymous method in one important way: it can return to one of two different places. The normal return route is to the method that activated the block by sending it the value message. However, control may also return to the method that activated the textually enclosing method (i.e., the method in which the block was bound). This is indicated by placing an uparrow before the last statement in a Smalltalk block. In the latter case, the activation of methods is not LIFO. The inclusion of this feature would complicate the semantics enormously, and so has not been covered in this limited exposition.

Context conditions stating exactly which programs are considered to be legal are to be found in the appendix.

4 The Object Model in Smalltalk

Smalltalk methods operate on objects which reside in a single, persistent store. This store, or object memory, contains all the objects that exist in a Smalltalk system, including methods, classes, and "primitive objects" such as integers. Note also that method activation records, or *contexts* in Smalltalk terminology, also reside in the object memory. However, for simplicity our semantics does not place contexts in the object store.

Object_memory = map *Oop* to *Object*

We shall usually denote values of *Object_memory* by σ.

Every object is identified by a unique internal name, or *object pointer* (here contracted to *Oop*). Every object is an instance of a class:

Object :: *Class* : *Class_name*
Body : *Object_body*

[1] The type *Ulist*(*X*) models a list with no duplicate elements. Formally, *Ulist*(*X*) = seq of *X*, where *inv-Ulist*(*X*)(*l*) $\triangleq$ len *l* = card rng *l*.

[2] Actually, Smalltalk does not allow blocks to have temporaries. The absence of temporary variables from blocks was a curious omission in the design of Smalltalk. Later we shall meet other strange features of blocks.

$Object\_body = Plain\_object \cup Primitive\_object$

$Primitive\_object = \mathbf{Z} \cup Block$

Primitive objects have no instance variables and are therefore immutable: their internal state cannot change.[3]

Plain_objects are instances of user-defined classes. They contain references to other objects. In Smalltalk, the instance variables of an object are exactly those declared in its class and superclasses (except for indexed instance variables). This leads naturally to the following model of *Plain_objects*:

$Plain\_object = \text{map } Id \text{ to } Oop$

Smalltalk implementations linearise the instance variables, so that each is assigned a unique integer, leading to the following model:

$Plain\_object = \text{seq of } Oop$

However, this creates severe problems when multiple inheritance is involved. For the moment, we shall choose the former model, and introduce indexed instance variables later.

5 Environments

In the semantic equations that follow we use two structures that we term "environments". The *static environment* describes the textual context of a method with respect to the rest of the program. This is constant for any particular method within a program. The *dynamic environment* describes the local state of an invocation of a method (it is anologous to a Smalltalk MethodContext). It changes during the invocation of a method, and each invocation has its own local dynamic environment. In addition to these there is the object memory which is global. One can consider these to be, from an operational point of view, the environment known at compile-time, the local values known at method invocation time, and the store, respectively.

5.1 The Static Environment

A static environment, denoted by ρ, contains the "text" of all the methods, and an indication of which class the current method is in (required by the super mechanism):

$$\begin{aligned} SEnv :: \; & Class : Class\_name \\ & P : Program \end{aligned}$$

5.2 The Dynamic Environment

A dynamic environment, denoted by δ, records the state of the computation local to a method invocation:

$$\begin{aligned} DEnv :: \; & Rcvr : Oop \\ & Args : \text{map } Id \text{ to } Oop \\ & Temps : \text{map } Id \text{ to } Oop \end{aligned}$$

The receiver and arguments of a method are determined when a method is bound to a message, and do not change during the execution of the method. Temporaries, however, are initialised to nil and are usually assigned to within the method.

6 The Semantic Function for Methods

In our semantics, methods and blocks are functions which transform the object memory. In reality, they are encoded into CompiledMethods, which are first-class objects interpreted by the Smalltalk Virtual Machine. As mentioned earlier, this has been ignored in the existing semantics, and methods no longer reside in the *Object_memory* as full objects.

A *Method* takes a receiver and a list of arguments, and returns a result object, transforming the object memory as a side-effect:

$Method = Oop \times \text{seq of } Oop \times Object\_memory \rightarrow Oop \times Object\_memory$

[3] Most Smalltalk implementations take advantage of this by encoding the value of an integer object into its object pointer.

A *Block* is similar, but the associated receiver is the block-object (see later section on blocks).

$Block = Oop \times \text{seq of } Oop \times Object\_memory \to Oop \times Object\_memory$

An initial object memory will contain not only the methods and blocks specified by the Smalltalk program, but also so-called "primitive" methods, which cannot be expressed in Smalltalk[2]. Such primitive methods include those for integer arithmetic and comparison, and special methods for activating blocks. This is why the *Methods* field of a *Class_body* can map a *Selector* to a "pre-compiled" method, known as a *Primitive_method*.

$Primitive\_method = Method$

We now present the semantic function for methods. It takes a method definition, in its static environment, and returns a method denotation.

$$MMethod\_body : Method\_body \to SEnv \to Method$$

$$MMethod\_body[\![mk\text{-}Method\_body(args, temps, exprs)]\!]\rho \triangleq$$
$$\lambda rcvr, arglist, \sigma \cdot$$
$$\quad \text{let } \delta = mk\text{-}DEnv(rcvr, \{args(i) \mapsto arglist(i) \mid i \in \text{dom } args\}, \{id \mapsto \text{NILOOP} \mid id \in temps\}) \text{ in}$$
$$\quad \text{let } (result, \delta', \sigma') = MExpression\_list[\![exprs]\!]\rho\delta\sigma \text{ in}$$
$$\quad (result, \sigma')$$

The result value of a list of expressions is the result of the last expression in the list.

$$MExpression\_list : Expression\_list \to SEnv \to DEnv \to Object\_memory$$
$$\to Oop \times DEnv \times Object\_memory$$
$$MExpression\_list[\![exprs]\!]\rho\delta\sigma \triangleq$$
$$\quad \text{let } (oop, \delta', \sigma') = MExpression[\![\text{hd } exprs]\!]\rho\delta\sigma \text{ in}$$
$$\quad \text{if len } exprs = 1 \text{ then } (oop, \delta', \sigma') \text{ else } MExpression\_list[\![\text{tl } exprs]\!]\rho\delta'\sigma'$$

7 The Semantic Function for Expressions

7.1 Assignment

Expressions change the local state of the computation by altering the dynamic environment. An assignment can only alter the values of temporaries or instance variables; arguments are read-only.

$$MExpression : Expression \to SEnv \to DEnv \to Object\_memory \to Oop \times DEnv \times Object\_memory$$

$$MExpression[\![mk\text{-}Assignment(id, rhs)]\!]\rho\delta\sigma \triangleq$$
$$\quad \text{let } (result, \delta', \sigma') = MExpression[\![rhs]\!]\rho\delta\sigma \text{ in}$$
$$\quad \text{if } id \in \text{dom } Temps(\delta)$$
$$\quad \text{then } (result, \mu(\delta', Temps \mapsto Temps(\delta') \dagger \{id \mapsto result\}), \sigma')$$
$$\quad \text{else } (result, \delta', \sigma' \dagger \{Rcvr(\delta) \mapsto \mu(\sigma(Rcvr(\delta)), Body \mapsto Body(\sigma(Rcvr(\delta))) \dagger \{id \mapsto result\})\})$$

7.2 Variables

$$MExpression[\![id]\!]\rho\delta\sigma \triangleq \text{if } id \in \text{dom } Temps(\delta)$$
$$\text{then } (Temps(\delta)(id), \delta, \sigma)$$
$$\text{else if } id \in \text{dom } Args(\delta)$$
$$\text{then } (Args(\delta)(id), \delta, \sigma)$$
$$\text{else } \Big(Body(\sigma(Rcvr(\delta)))(id), \delta, \sigma\Big)$$

7.3 Pseudo-variables

$$MExpression[\![\text{SELF}]\!]\rho\delta\sigma \triangleq (Rcvr(\delta), \delta, \sigma)$$

$MExpression[\![\text{SUPER}]\!]\rho\delta\sigma \triangleq (Rcvr(\delta), \delta, \sigma)$

ROOT is present in all object memories, and therefore has a constant *Oop*, ROOTOOP.

$MExpression[\![\text{ROOT}]\!]\rho\delta\sigma \triangleq (\text{ROOTOOP}, \delta, \sigma)$

7.4 Integer Constants

$MExpression[\![int]\!]\rho\delta\sigma \triangleq$
  let $(oop, \sigma') = find\_or\_make\_int(int, \sigma)$ in
  (oop, δ, σ') for $int \in \mathbf{Z}$

The *find_or_make_int* function, given an integer and an object memory, returns the *Oop* of an integer object that has the integer as its value. It is specified as a VDM operation in such a way that duplicate integer objects are allowed (to give an implementor freedom).

find_or_make_int $(int: \mathbf{Z})$ $obj: Oop$
ext wr σ : *Object_memory*
post $\sigma(obj) = mk\text{-}Object(\text{Integer}, int) \wedge (\sigma = \overleftarrow{\sigma} \vee \sigma \triangleleft \{obj\} = \overleftarrow{\sigma})$

7.5 Message Sending

Sending messages is the only way in Smalltalk for objects to communicate with each other. Smalltalk messages are synchronous: the sender waits for the receiver to return a value before continuing. A Smalltalk message can therefore be considered to be a dynamically bound procedure call; the particular method to be executed in response to a message is determined by the class of the receiver. When a message is received by an object a search is performed for a corresponding method, starting at the class of the receiver, and working up through its superclasses.

Additionally, in Smalltalk there is an alternative starting point for the search, indicated by an object sending a message to itself with the special designation super. This starts the search in the superclass of the class in which the message is being sent from. This allows a class to override the behaviour of an inherited method but still have access to that method.

In the formal semantics, these searches are distinguished by different bindings of the *find* function. The super search is based entirely on the static environment, and can therefore be bound at compile-time, whereas the usual search cannot.

$MExpression[\![mk\text{-}Message\_list(rcvr, msgs)]\!]\rho\delta\sigma \triangleq$
  let $(rcvr\_object, \delta', \sigma') = MExpression[\![rcvr]\!]\rho\delta\sigma$ in
  let $search\_fn =$ if $rcvr = \text{SUPER}$
    then $find(Super(P(\rho)(Class(\rho))), \rho)$
    else $find(Class(\sigma'(rcvr\_object)), \rho)$
  in
  $MMessage\_list[\![msgs]\!]\rho\delta'(search\_fn, rcvr\_object, \sigma')$

The method search may fail to find a method corresponding to a particular message selector, in which case the search function returns nil (i.e., a VDM nil, not to be confused with the Smalltalk nil object).

$Search\_function = Selector \to [Method]$

$find : [Class\_name] \times SEnv \to Search\_function$

$find(class, \rho)sel \triangleq$
  if $class = \text{nil}$ then nil else if $sel \in \text{dom}\, Methods(P(\rho)(class))$
    then if $Methods(P(\rho)(class))(sel) \in Primitive\_method$
      then $Methods(P(\rho)(class))(sel)$
      else $MMethod\_body[\![Methods(P(\rho)(class))(sel)]\!]\mu(\rho, Class \mapsto class)$
    else $find(Super(P(\rho)(class)), \rho)sel$

Sending a list of messages to an object consists of evaluating the arguments of the first message, sending the first message, then sending the rest of the messages in the list.

$$\begin{aligned}
&\mathit{MMessage_list} : \text{seq of } \mathit{Message} \to \mathit{SEnv} \to \mathit{DEnv} \to (\mathit{Search_function} \times \mathit{Oop} \times \mathit{Object_memory}) \\
&\qquad \to (\mathit{Oop} \times \mathit{DEnv} \times \mathit{Object_memory}) \\
&\mathit{MMessage_list}[\![\mathit{msgs}]\!]\rho\delta(\mathit{search_fn}, \mathit{rcvr}, \sigma) \triangleq \\
&\quad \text{let } (v, \delta', \sigma') = \mathit{MMessage}[\![\text{hd } \mathit{msgs}]\!]\rho\delta(\mathit{search_fn}, \mathit{rcvr}, \sigma) \text{ in} \\
&\quad \text{if len } \mathit{msgs} = 1 \text{ then } (v, \delta', \sigma') \text{ else } \mathit{MMessage_list}[\![\text{tl } \mathit{msgs}]\!]\rho\delta'(\mathit{search_fn}, \mathit{rcvr}, \sigma')
\end{aligned}$$

$$\begin{aligned}
&\mathit{MMessage} : \mathit{Message} \to \mathit{SEnv} \to \mathit{DEnv} \\
&\qquad \to (\mathit{Search_function} \times \mathit{Oop} \times \mathit{Object_Memory}) \to (\mathit{Oop} \times \mathit{DEnv} \times \mathit{Object_Memory}) \\
&\mathit{MMessage}[\![\mathit{mk\text{-}Message}(\mathit{sel}, \mathit{arglist})]\!]\rho\delta(\mathit{search_fn}, \mathit{rcvr}, \sigma) \triangleq \\
&\quad \text{let } (\mathit{actuals}, \delta', \sigma') = \mathit{MAll_Expression_list}[\![\mathit{arglist}]\!]\rho\delta\sigma \text{ in} \\
&\quad \text{let } (\mathit{result}, \sigma'') = \mathit{perform}(\mathit{rcvr}, \mathit{sel}, \mathit{actuals}, \mathit{search_fn}, \sigma') \text{ in} \\
&\quad (\mathit{result}, \delta', \sigma'')
\end{aligned}$$

MAll_Expression_list evaluates a list of expressions, returning a list of results.

$$\begin{aligned}
&\mathit{MAll_Expression_list} : \text{seq of } \mathit{Expression} \to \mathit{SEnv} \to \mathit{DEnv} \to \mathit{Object_Memory} \\
&\qquad \to \text{seq of } \mathit{Oop} \times \mathit{DEnv} \times \mathit{Object_Memory} \\
&\mathit{MAll_Expression_list}[\![\mathit{el}]\!]\rho\delta\sigma \triangleq \\
&\quad \text{if } \mathit{el} = [\,] \\
&\quad \text{then } ([\,], \delta, \sigma) \\
&\quad \text{else let } (\mathit{val}, \delta', \sigma') = \mathit{MExpression}[\![\text{hd } \mathit{el}]\!]\rho\delta\sigma \text{ in} \\
&\qquad \text{let } (\mathit{val_list}, \delta'', \sigma'') = \mathit{MAll_Expression_list}[\![\text{tl } \mathit{el}]\!]\rho\delta'\sigma' \text{ in} \\
&\qquad ([\mathit{val}] \frown \mathit{val_list}, \delta'', \sigma'')
\end{aligned}$$

If a particular method search fails, the special message doesNotUnderstand: must be sent to the receiver. A method corresponding to the doesNotUnderstand: message is searched for in the same way. This allows a class to override the behaviour of doesNotUnderstand:.

$$\begin{aligned}
&\mathit{perform} : \mathit{Oop} \times \mathit{Selector} \times \text{seq of } \mathit{Oop} \times \mathit{Search_function} \times \mathit{Object_memory} \to \mathit{Oop} \times \mathit{Object_Memory} \\
&\mathit{perform}(\mathit{rcvr}, \mathit{sel}, \mathit{args}, \mathit{search_fn}, \sigma) \triangleq \\
&\quad \text{let } \mathit{first_search_result} = \mathit{search_fn}(\mathit{sel}) \text{ in} \\
&\quad \text{if } \mathit{first_search_result} \neq \text{nil} \\
&\quad \text{then } \mathit{first_search_result}(\mathit{rcvr}, \mathit{args}, \sigma) \\
&\quad \text{else let } \mathit{second_search_result} = \mathit{search_fn}(\text{doesNotUnderstand:}) \text{ in} \\
&\qquad \text{let } (\mathit{message}, \sigma') = \mathit{create_message}(\mathit{sel}, \mathit{args}, \sigma) \text{ in} \\
&\qquad \mathit{second_search_result}(\mathit{rcvr}, [\mathit{message}], \sigma')
\end{aligned}$$

Note that for the result of the *perform* function to be defined should the original message not be understood, there *must* exist a method corresponding to the doesNotUnderstand: selector.

The *create_message* function creates an instance of class **Message** that records the selector and arguments of the message that was not understood.

7.6 Blocks

The implementation of blocks in Smalltalk does not work as one might expect. When a block is bound to its surrounding context, an instance of BlockContext is created that records the binding. The BlockContext also has memory reserved for the workspace required when the block is activated. Unfortunately, blocks can be activated re-entrantly. This means that most Smalltalk implementations fail in unexpected ways when presented with re-entrant blocks, because the workspace of one activation is overwritten by another. For example, the following Smalltalk code, which looks reasonable at first sight, would not work on most systems:

```
factBlock ← [ :n |
  n=0 ifTrue: [1]
      ifFalse: [n * (factBlock value: n−1)]].
factBlock value: 2
```

The solution to the problem is to make a clean separation between the acts of binding the block to its surrounding context, and activating a bound block. To this end, we propose that binding a block create an instance of a new class, Closure. The Closure would be a first-class object. To activate the block, the special value message would be sent to the Closure. This leads to the following semantics:

$$MExpression[\![block]\!]\rho\delta\sigma \triangleq$$
$$\quad \text{let } (closure, \sigma') = make\_closure(block, \rho, \delta, \sigma) \text{ in}$$
$$\quad (closure, \delta, \sigma') \qquad \text{for } block \in Block\_body$$

When the closure is created the block is bound to the dynamic environment in existence at the time.

$$make\_closure \ (block\text{: } Block\_body, \rho\text{: } SEnv, \delta\text{: } DEnv) \ closure\text{: } Oop$$
$$\text{ext wr } \sigma \ : \ Object\_Memory$$
$$\text{post } \sigma(closure) = mk\text{-}Object(\text{Closure}, MBlock\_body[\![block]\!]\rho\delta) \wedge (\sigma = \overleftarrow{\sigma} \vee \sigma \lhd \{closure\} = \overleftarrow{\sigma})$$

As mentioned earlier, a block is bound in such a way that it has access to the receiver of the method in which it was bound, and the arguments of the enclosing method (and any enclosing blocks) but not to the temporary variables of enclosing methods/blocks. Access to these would require dynamic environments to outlive their associated invocations, complicating the model somewhat.

Note also that a block must check at run-time that it was invoked with the same number of arguments that it had in its definition. If this is not the case, an error message is sent.

$$MBlock\_body : Block\_body \to SEnv \to DEnv \to Block$$

$$MBlock\_body[\![mk\text{-}Block\_body(args, temps, exprs)]\!]\rho\delta(closure, arglist, \sigma) \triangleq$$
$$\quad \text{let } (result, \delta'', \sigma') =$$
$$\quad\quad \text{if len } arglist = \text{len } args$$
$$\quad\quad \text{then let } \delta' = mk\text{-}DEnv(Rcvr(\delta), Args(\delta) \cup \{args(i) \mapsto arglist(i) \mid i \in \text{dom } args\},$$
$$\quad\quad\quad\quad \{id \mapsto \text{NILOOP} \mid id \in temps\}) \text{ in}$$
$$\quad\quad\quad MExpression\_list[\![exprs]\!]\rho\delta'\sigma$$
$$\quad\quad \text{else } MMessage[\![mk\_Message(mk\text{-}Unary(\text{wrongNumberOfArguments}), [\,])]\!]\rho\delta$$
$$\quad\quad\quad\quad (find(Class(\sigma(closure)), \rho), closure, \sigma)$$
$$\quad\quad \text{in}$$
$$\quad (result, \sigma')$$

8 Primitives

Every Smalltalk system has to have a complement of primitive operations provided. In this section, we outline how the semantics of such primitives can be described within our model. There are far too many primitives in Smalltalk to cover here, but many are provided purely for efficiency purposes, and many are similar to each other. We will only describe a few: the value: primitive for activating blocks, the new primitive for creating objects, an arithmetic primitive, and the perform: primitive for computed selectors.

8.1 The value: primitive

The value: primitive activates a block with one argument. The other forms of value for zero, and two or more arguments would be similar.

$$value\_primitive : Method$$
$$value\_primitive \triangleq \lambda closure, [arg], \sigma \cdot Body(\sigma(closure))(closure, [arg], \sigma)$$

8.2 The new primitive

Up to now we have not dealt with the fact that in Smalltalk, classes are objects. We shall simply state that an initial object memory should contain representations of the classes in the static environment, and that a function is required to map the representation of the class to its abstract syntax. The operation presented here, which creates a new instance of a class, is passed the abstract syntax of the class as an argument.

new_primitive (*class*: *Class_name*) *new_oop*: *Oop*
ext wr σ : *Object_Memory*
rd P : *Program*
post $new\_oop \notin \mathrm{dom}\ \overleftarrow{\sigma}$
$\wedge\ \sigma = \overleftarrow{\sigma} \cup \{new\_oop \mapsto mk\text{-}Object(class, \{id \mapsto \textsc{NilOop} \mid id \in inst\_vars(class, P)\})\}$

A definition of *inst_vars* can be found in the appendix.

8.3 An Arithmetic Primitive

As an example of one of the many arithmetic primitives, we present the primitive for integer addition. Rather than model the Smalltalk Virtual Machine concept of primitive failure[2], we choose to return nil if the argument to the primitive is not an Integer.

plus_primitive : *Method*
plus_primitive $\triangleq$
$\lambda rcvr, [arg], \sigma \cdot$
let *addend* = $Body(\sigma(rcvr))$ in
let *augend* = $Body(\sigma(arg))$ in
if *augend* $\in \mathbf{Z}$
then *find_or_make_int*(*addend* + *augend*, σ)
else ($\textsc{NilOop}, \sigma$)

8.4 The perform: Primitive

The perform: primitive takes a Symbol representing a selector, and sends a message to an object using that selector as the name of the message. As with the new primitive, we have not explicitly stated that selectors have representations in the object memory, but we assume that a mechanism exists for deriving the abstract syntax of the selector from its stored representation. The following operation assumes that its selector is a unary selector.

perform_primitive (*obj*: *Oop*, *sel*: *Selector*) *result*: *Oop*
ext wr σ : *Object_Memory*
rd ρ : *SEnv*
post $(result, \sigma) = perform(obj, sel, [\,], find(Class(\overleftarrow{\sigma}(obj)), \rho), \overleftarrow{\sigma})$

9 Indexed Instance Variables

Thus far we have not mentioned indexed instance variables. We now outline how the model can be adapted to include indexed instance variables.

Firstly, a class can either define that all its instances may have indexed instance variables, or inherit the property from its superclass. This requires the addition of a field, *Has_indexed*: **B**, to the definition of *Class_body*. One can then determine whether a class' instances may have indexed instance variables with the following function:

has_indexed : *Class_name* $\times$ *Class_map* $\rightarrow$ **B**
has_indexed(*class*, *class_map*) $\triangleq$ *Has_indexed*(*class_map*(*class*))
$\vee$ (*Super*(*class_map*(*class*)) $\neq$ nil $\Rightarrow$ *has_indexed*(*Super*(*class_map*(*class*)), *class_map*))

Secondly, the definition of *Plain_object* has to be modified to include the integer indices into the domain of the map:

$Plain\_object = \text{map } Id \cup \mathbf{N}_1 \text{ to } Oop$

Next, the new: primitive is provided so that objects with indexed instance variables may be created:

$new:\_primitive \ \ (class: Class\_name, size: \mathbf{N}_1)\ new\_oop: Oop$

ext wr σ : $Object\_memory$
rd P : $Program$

pre $has\_indexed(class, P)$

post $new\_oop \notin \text{dom } \overleftarrow{\sigma}$
$\wedge\ (\sigma = \overleftarrow{\sigma} \cup mk\text{-}Object(class, \{name \mapsto \text{NILOOP} \mid name \in instvars(class, P) \cup \{1, \ldots, size\}\}))$

Finally, the at: and at:put: primitives have to be provided to allow access to the indexed instance variables:

$at\_primitive : Method$

$at\_primitive \ \triangleq\ \lambda obj, [index], \sigma \cdot \text{if } in\_bounds(obj, index, \sigma)$
$\text{then } (Body(\sigma(obj))(index), \sigma)$
$\text{else } bound\_error(obj, index, \sigma)$

$atput\_primitive : Method$

$atput\_primitive \ \triangleq$
$\lambda obj, [index, value], \sigma \cdot$
$\text{if } in\_bounds(obj, index, \sigma)$
$\text{then } (value, \sigma \dagger \{obj \mapsto \mu(\sigma(obj), Body \mapsto Body(\sigma(obj)) \dagger \{index \mapsto value\})\})$
$\text{else } bound\_error(obj, index, \sigma)$

$in\_bounds : Oop \times Oop \times Object\_Memory \to \mathbf{B}$

$in\_bounds(obj, index, \sigma) \ \triangleq\ Body(\sigma(index)) \in \text{dom } Body(\sigma(obj))$

The *bound_error* function sends an appropriate error message; it will not be defined further here.

10 Discussion

We have presented semantics for most of the Smalltalk language. A number of things have been omitted:

1. Concurrency has been omitted because it would severely complicate the existing model. In fact, it is doubtful whether the existing approach can deal satisfactorily with concurrent processes at all.

2. Classes, selectors and methods do not appear as objects in the object memory; this is purely to make the model more understandable.

3. Blocks are not allowed to perform non-local returns and do not have access to non-local temporaries. Introducing non-local returns requires the use of continuation semantics[3,6] to carry around the extra possible return path. It also introduces a class of programming error whereby one can return to a context that has already been returned from.

Allowing access to non-local temporaries requires a more sophisticated mechanism to release contexts. In the Smalltalk-80 system each context is a full object, and because a method context can be referenced by a block context, it cannot be released automatically upon return. To keep the semantics simple we avoided non-local references to temporaries. However, they can be done away with quite easily by adding an indirection object to methods that contain blocks that access a non-local variable, and changing all references to the temporary to go via the indirection object instead. For example,

```
| i |
i ← 1.
[i < 10] whileTrue: [i ← i + 1]
```

can be rewritten as:

```
| i |
i ← Indirection on: 1.
[i value < 10] whileTrue: [i value: i value + 1].
```

where the following Indirection class is assumed to exist:

class	Indirection
superclass	Object
instance variables	object

class methods

```
on: anObject
↑self new value: anObject
```

instance methods

```
value
↑object

value: anObject
↑object ← anObject
```

In this case, the value of i accessible to the blocks in the **whileTrue:** statement is determined when the blocks are bound; essentially it is an extra argument to the blocks. The semantics are changed by altering the creation of the dynamic environment in *MBlock_body* to use the following expression:

$$\delta' = \mathit{mk\text{-}DEnv}(\mathit{Rcvr}(\delta), \mathit{Args}(\delta) \cup \mathit{Temps}(\delta) \cup \{\mathit{Args}(i) \mapsto \mathit{arglist}(i) \mid i \in \mathrm{dom}\ \mathit{args}\}, \\ \{\mathit{id} \mapsto \textsc{NilOop} \mid \mathit{id} \in \mathit{temps}\})$$

The context condition for blocks (see Appendix) is similarly modified so that non-local temporaries appear as arguments.

This transformation, together with the removal of non-local returns enables us to have LIFO contexts. The implications for implementations of the Smalltalk Virtual Machine are large, and may lead to significant performance gains. However, whether the loss of non-local returns significantly hampers programming style has yet to be determined. Clearly, the loss of access to non-local temporaries is not a problem; a pre-processor could mechanically transform existing code to use indirection objects.

11 Conclusions

The semantics of a large subset of Smalltalk have been presented. It is believed that discussion of language features is greatly aided by basing such discussion on formal semantics. Further work has to be done to cope with more of the language (parallelism, multiple inheritance); this work is the subject of current research. From the viewpoint of the formal semantics presented here it is felt that a better understanding of Smalltalk and other object-oriented languages will be gained, leading to improved designs in the future.

12 Acknowledgements

Numerous discussions with Cliff Jones helped shape the ideas in this paper. Many thanks to Michael Fisher for his careful reading and useful suggestions.

References

[1] D. Bjørner and C. B. Jones. *Formal Specification and Software Development.* Prentice-Hall, 1983.

[2] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

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

[4] C. B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall International, 1986.

[5] C. Minkowitz and P. Henderson. A formal description of object-oriented programming using VDM. In D. Bjørner, C. B. Jones, M. M. an Airchinnigh, and E. J. Neuhold, editors, *VDM '87: VDM—A Formal Method at Work*, pages 237–259, Springer-Verlag, Brussels, Belgium, March 1987.

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

[7] M. Wolczko. *Specification and Implementation of Object-Oriented Systems (working title).* PhD thesis, Department of Computer Science, University of Manchester, forthcoming.

A Context Conditions

In this appendix we state the context conditions defining which programs are well-formed.

The static environment required by the context conditions simply records which identifiers are in scope, and what sort of variable they denote.

$$CCEnv = \mathsf{map}\ Id\ \mathsf{to}\ Var\_Type$$

$$Var\_Type = \{\textsc{InstVar}, \textsc{Arg}, \textsc{Temp}\}$$

A *Program* is well-formed if the graph of classes is well-founded, and each individual class is well-formed with respect to its superclasses:

$$WFProgram : Program \to \mathbf{B}$$

$$\begin{aligned} &WFProgram[\![class\_map]\!] \triangleq non\_circular(class\_map) \\ &\quad \wedge \forall class\_id \in \mathsf{dom}\ class\_map \cdot \\ &\qquad Super(class\_map(class\_id)) \in \mathsf{dom}\ class\_map \cup \{\mathsf{nil}\} \wedge \\ &\qquad \mathsf{let}\ inh\_iv = inherited\_inst\_vars(class\_id, class\_map)\ \mathsf{in} \\ &\qquad WFClass[\![class\_map(class\_id)]\!]\{id \mapsto \textsc{InstVar} \mid id \in inh\_iv\} \end{aligned}$$

$$non\_circular : Class\_map \to \mathbf{B}$$

$$non\_circular(class\_map) \triangleq \forall s \subseteq \mathsf{dom}\ class\_map \cdot s \neq \{\} \Rightarrow \exists class \in s \cdot Super(class\_map(class)) \notin s$$

$$inherited\_inst\_vars : Class\_name \times Class\_map \to \mathsf{set\ of}\ Id$$

$$\begin{aligned} &inherited\_inst\_vars(class, class\_map) \triangleq \\ &\quad \mathsf{if}\ Super(class\_map(class)) = \mathsf{nil}\ \mathsf{then}\ \{\}\ \mathsf{else}\ inst\_vars(Super(class\_map(class)), class\_map) \end{aligned}$$

$$inst\_vars : Class\_name \times Class\_map \to \mathsf{set\ of}\ Id$$

$$inst\_vars(class, class\_map) \triangleq Instvars(class\_map(class)) \cup inherited\_inst\_vars(class, class\_map)$$

A class is well-formed if it does not redeclare any inherited instance variables, and all its methods are well-formed.

$$WFClass : Class\_body \to CCEnv \to \mathbf{B}$$

$$\begin{aligned} &WFClass[\![mk\text{-}Class\_body(iv, super, meths)]\!]\theta \triangleq is\text{-}disjoint(iv, \mathsf{dom}\,\theta) \\ &\quad \wedge \forall sel \in \mathsf{dom}\ meths \cdot \\ &\qquad meths(sel) \notin Primitive\_method \\ &\qquad\quad \Rightarrow nargs(sel) = \mathsf{len}\ Args(meths(sel)) \\ &\qquad\qquad \wedge WFMethod[\![meths(sel)]\!](\theta \cup \{id \mapsto \textsc{InstVar} \mid id \in iv\}) \end{aligned}$$

A method is well-formed if none of its instance variables, arguments or temporaries are multiply declared, if it has at least one expression,[4] and all its expressions are well formed.

$$WFMethod : Method\_body \to CCEnv \to \mathbf{B}$$

$$WFMethod[\![mk\text{-}Method\_body(args, temps, exprs)]\!]\theta \triangleq all\_disjoint([\mathsf{dom}\,\theta, \mathsf{rng}\,args, temps])$$
$$\wedge\ \mathsf{len}\,exprs \geq 1$$
$$\wedge\ \forall e \in \mathsf{rng}\,exprs \cdot$$
$$WFExpression[\![e]\!](\theta \cup \{id \mapsto \textsc{Arg} \mid id \in \mathsf{rng}\,args\} \cup \{id \mapsto \textsc{Temp} \mid id \in temps\})$$

$$all\_disjoint : \mathsf{seq\ of\ set\ of}\ Id \to \mathbf{B}$$

$$all\_disjoint(ss) \triangleq \forall i, j \in \mathsf{dom}\,ss \cdot i \neq j \Rightarrow is\_disjoint(ss(i), ss(j))$$

Now we shall enumerate by cases the well-formedness condition for each type of *Expression*.

An assignment is well-formed if it assigns to a variable (not an argument) that is in scope, and its RHS is well-formed.

$$WFExpression : Expression \to CCEnv \to \mathbf{B}$$

$$WFExpression[\![mk\text{-}Assignment(id, rhs)]\!]\theta \triangleq$$
$$WFExpression[\![rhs]\!]\theta \wedge id \in \mathsf{dom}\,\theta \wedge \theta(id) \in \{\textsc{InstVar}, \textsc{Temp}\}$$

$$WFExpression[\![id]\!]\theta \triangleq id \in \mathsf{dom}\,\theta \quad \text{for } id \in Id$$

$$WFExpression[\![\textsc{Self}]\!]\theta \triangleq \mathsf{true}$$

$$WFExpression[\![\textsc{Super}]\!]\theta \triangleq \mathsf{true}$$

$$WFExpression[\![\textsc{Root}]\!]\theta \triangleq \mathsf{true}$$

$$WFExpression[\![mk\text{-}Message\_list(rcvr, msgs)]\!]\theta \triangleq WFExpression[\![rcvr]\!]\theta \wedge \mathsf{len}\,msgs \geq 1$$
$$\wedge\ \forall m \in \mathsf{rng}\,msgs \cdot$$
$$\mathsf{len}\,Args(m) = nargs(Sel(m)) \wedge \forall arg \in \mathsf{rng}\,Args(m) \cdot WFExpression[\![arg]\!]\theta$$

$$WFExpression[\![int]\!]\theta \triangleq \mathsf{true} \quad \text{for } int \in \mathbf{Z}$$

$$WFExpression[\![mk\text{-}Block\_body(args, temps, exprs)]\!]\theta \triangleq$$
$$WFMethod[\![mk\text{-}Method\_body(args, temps, exprs)]\!](\theta \rhd \{\textsc{InstVar}, \textsc{Arg}\})$$

If, as discussed in the text, we allow a block to have read-only access to non-local temporaries, the definition becomes:

$$WFExpression[\![mk\text{-}Block\_body(args, temps, exprs)]\!]\theta \triangleq$$
$$WFMethod[\![mk\text{-}Method\_body(args, temps, exprs)]\!](\theta \rhd \{\textsc{InstVar}, \textsc{Arg}\})$$
$$\cup \{id \mapsto \textsc{Arg} \mid id \in \mathsf{dom}\,\theta \wedge \theta(id) = \textsc{Temp}\}$$

[4] Smalltalk *does* allow empty methods and empty blocks. However, the former are equivalent to ↑self, and the latter return nil. We therefore insist that all methods and blocks have at least one expression.