Abstract

An operational semantics for Smalltalk is presented. While the description is not very interesting in itself, it does have a couple of merits.

First, ambiguities in existing Smalltalk systems (such as the value of 5+nil) can be resolved. Second, we are able to give a simple model for the complicated exit-semantics of Smalltalk blocks. Third, the semantics can be used as a basis for “collecting semantics” in analysis and optimization algorithms.

Of these, we consider the analysis aspect most important and there is already work in progress on type inference and closure analysis, based on the operational semantics given here. We present some of these ideas at the end of this paper.

1 Introduction

In the early 70’s, Smalltalk pioneered the field of object-oriented programming [GR83, LP90]. Its success can be attributed not only to the object-oriented paradigm itself, but also to the simplistic and general nature of the language; in Smalltalk, everything is an object and message passing is the only control structure. Blocks (≈ λ-expressions) are used to model even the simplest control structures, but are flexible enough to achieve, and exceed, the power of iterators as offered by other languages.

More than a decade later, C++ gained a great deal of popularity by using the same abstraction mechanism as Smalltalk, but also by capitalizing on the efficiency concerns of the C community. Today, many of Smalltalk’s ideas are finding its way into newer languages, Java being the primary example. Also, Smalltalk is gaining more acceptance and wider use.

Naturally, most programmers would rather leave administrative details to the compiler, and not have to
worry about *e.g.* memory management and type declarations. However, a language as general as Smalltalk requires a lot of analysis and optimization on behalf of the compiler. Quite often, such analysis is easier to formulate and prove correct when based on the exact semantics of the language, as in abstract interpretation [CC77].

This work originated two years ago when the authors began to formulate a type inference algorithm for the Smalltalk dialect RTT [EMTP+96]. RTT is a programming language for real-time applications whose syntax and semantics is based on Smalltalk, with some limitations pertinent to real-time systems.

For correctness and simplicity we wished to let the algorithm use a collecting semantics, based on an operational definition of Smalltalk. Structural operational semantics [NN92] offers sufficient detail and expressiveness for our purposes, but, at the time, there was no operational definition available \(^1\). Hence this paper. There are also denotational descriptions of Smalltalk, see for instance Kamin’s description of inheritance [Kam88] and Golubski’s and Lippe’s description of Smalltalk-80 [GL95].

In the course of exploring Smalltalk, we found a few discrepancies between existing Smalltalk systems. For example, one Smalltalk system gave an error message when presented with the expression 5+nil, while another returned 5. One system complained if not all arguments to a block were given upon invocation while another went right ahead and executed the block, with missing arguments set to nil. While it is not our intention to define a standard for Smalltalk in this paper, we hope that a forthcoming standard may benefit from this work.

To simplify the presentation, we first describe µtalk, a subset of Smalltalk without blocks (section 2). Since blocks are of interest mainly to Smalltalk, this separation has the added advantage of making the semantics of standard object-oriented constructs, like message passing, available to other languages.

In section 3.1, blocks are described and added to µtalk. Finally, in section 6 we suggest how other Smalltalk features like global variables and meta-classes could be incorporated.

As already mentioned, our main interest lies in the ability to express analysis and optimization algorithms as “collecting semantics” based on the operational semantics for Smalltalk. In section 4 we briefly describe on-going work in this direction.

2 µtalk

For the purposes of this paper, we recall some Smalltalk idiosyncracies. Additional conventions are pointed out as we go along.

\(^1\)The authors would like to make the reviewers aware of two references whose title may seem to have bearings on this article. Of these, Mlotkowski [Mlo96] is actually using evolving algebras, a different formalism, and Clark’s thesis [Cla96] was not yet available, according to Barry Fry at QMW.
A new class must be given a unique name. It inherits methods and instance variables from its parent class. Methods can be overridden, but additional instance variable names must not clash with inherited variables.

Temporary variables are initially \texttt{nil} which is the only instance of the class \texttt{UndefinedObject}. In a similar fashion, identifiers \texttt{true} and \texttt{false} refer to pre-installed instances of classes \texttt{True} and \texttt{False}, respectively. Pseudo-variables \texttt{self} and \texttt{super} denote the receiver of the message.

\section*{2.1 Syntax}

It is customary to define the semantics based on the abstract syntax of the language. In our case, the parser does some additional processing to simplify the semantic definitions. Specifically, compound expressions appearing as arguments are first assigned to temporary variables. Assignments $x := e$ and return expressions $\uparrow e$ are broken down into the computation of $e$ and then the actual assignment/return operation. As an example, \texttt{obj m:1+2} is converted to the list $1+2 :: \text{assign t :: send obj m: t :: } \varepsilon$.

Two different return symbols are used to distinguish the two different ways of doing a return in Smalltalk. A local return, $\uparrow_{\text{loc}}$, marks the return from a method. If a Smalltalk method has no explicit return expression, the value of \texttt{self} is returned. The parser enforces this by inserting \texttt{self :: $\uparrow_{\text{loc}}$} at the end of each method. Non-local return, $\uparrow_{\text{nonloc}}$, can only appear in blocks and are thus described in sec. 3.1.

A message sent to \texttt{super} invokes a method in a parent of the class that defines the method in which the super call occurs (not necessarily the parent class of the receiver). Thus, the invoked method can be determined statically by the parser and there is no need for method lookup.

In addition to the abovementioned processing, the parser also emits a global class table, containing a list of all classes in the system.

The abstract syntax for $\mu$talk is presented in fig. 1. A program consists of an optional declaration of temporary variables and a sequence of expressions. An expression is either a simple literal, like a variable or an integer constant, or the act of sending a message to such a literal (henceforth called the receiver). Apart from the receiver, a message expression also consists of a selector — either unary or a binary/keyword — and a list of arguments to match the arity of the selector.

As mentioned earlier, messages to \texttt{super} requires no method lookup at run time; to reflect the static nature of this construct, the abstract syntax \texttt{super c s \ell_1 \ldots \ell_m} is used to carry the class name $c$ and the selector $s$ which together uniquely determines the method intended.

An expression can also be a call to a primitive, with optional arguments. Primitives perform such operations as arithmetic, storage management, control, and I/O. Assignments and instance creation are special and not viewed as sending a message. Finally, $\uparrow_{\text{loc}}$ returns the value of the previous expression to the sender.
2.2 Semantics

Smalltalk uses pointer semantics for assignment so that after \( v_1 := v_2 \), variables \( v_1 \) and \( v_2 \) are aliased.

We distinguish between three types of variables: temporaries, arguments, and instance variables (although class names can be viewed as global variables). Note that in Smalltalk, methods are not allowed to assign to argument variables. However, messages to argument objects, causing their state to change, are permitted, and also requires pointer semantics for passing of arguments. Instance variables can not be distinguished by their identifier alone; rather, they belong to a specific instance.

For the following discussion, please refer to fig. 1. An environment \( \text{Env} \) maps variables onto unique locations \( 0, 1, \text{etc.} \). Different environments are used depending on the variable at hand: \( \text{AEnv} \) for argument variables, \( \text{TEnv} \) for temporary variables, and \( \text{IEnv} \) for instance variables. Instances (\( \text{INST} \)) are characterized by the class from which they were created, and a unique instance number to tell instances of the same class apart. Instance variables are then described by their identifier and the instance number of the instance they belong to.

The store (\( \text{Store} \)) maps a location onto its current contents. A value (\( \text{Value} \)) is either an instance or a \( \text{SmallInteger} \) constant. As mentioned earlier, \texttt{true, false, and nil} are pre-defined instances, and their names and values will be loaded into the initial store.

When a method is invoked, in response to a message, the evaluation context must be saved. Only the argument and temporary environment, together with the remaining expressions, needs to be saved (\( \text{Context} \)) since the other information is considered “global”. The dump (\( \text{Dump} \)) records a history of the evaluation context. The state (\( \text{State} \)) finally, is a tuple holding the information in which expressions are evaluated, so that \( \tau = (\rho_{ae}, \rho_{te}, \rho_{ie}, \omega, \sigma, \text{free}, \delta) \) is an example of a state, where \( \omega \) holds the value of the most recent expression, and \( \text{free} \) denotes the next available store location.

A class (\( \text{Class} \)) is described by its name, superclass, methods, and instance variables. The special symbol \( \text{NoClass} \) marks the end of the inheritance hierarchy. A method (\( \text{Method} \)) consists of its name \( (\text{i.e., selector}) \), argument variables, and its method body with temporary variables. A list of all classes in the system are kept in the parser-generated constant \( \text{classList} \).

2.3 Auxiliary Functions

A number of auxiliary functions are used in the semantic rules. First, “\( \otimes \)” is used to compose two environments; \( \text{getLoc} \) finds the location of a variable, which in case of an instance variable must be accessed through the pseudo-variable \( \text{self} \); \( \text{lookup} \) retrieves a variable’s value while \( \text{evalLit} \) evaluates literals to a value; \( \text{update} \) stores a new value into a variable’s location. Notice that only temporary and instance variables can be mutated, and that the argument environment is used only for finding the instance number of \( \text{self} \) when
instance variables are updated. Finally, classOf returns the class name of a value, findMethod retrieves the method corresponding to the selector and the receiver’s class, and allInstVars yields all instance variables, including those inherited, available to a class.

2.4 Semantic Rules

The meaning of a program $p$ is defined by a transition system over a set of configurations, $\gamma$. A configuration is either intermediate, $\gamma = \langle p, \tau \rangle$, or final, $\gamma = \tau$. The transition relation “$\Rightarrow$” is defined using rules of structural operational semantics. The semantics of a program is defined transitively in terms of “$\Rightarrow$” so that

$$P[p, \tau_0] = \begin{cases} \tau & \text{if } \langle p, \tau_0 \rangle \Rightarrow \tau \\ \text{undef} & \text{otherwise.} \end{cases}$$

For the initial state $\tau_0$ (fig. 2) a program $p$ terminates iff there is a sequence of configurations $\gamma_0 = \langle p, \tau_0 \rangle \Rightarrow \gamma_1 \Rightarrow \ldots \Rightarrow \gamma_k$ such that $k$ is finite. If $\gamma_k$ is a final configuration, the body has terminated successfully. If $k$ is unbound, the program loops.

The rules to be presented are disjoint. Errors, such as “method not found”, are handled by making sure that none of the rules apply. For erroneous programs, the derivation stops in an intermediate configuration, suggesting that the input has no semantic meaning.

Refer to fig. 2 for the following explanation of the semantic rules for $\mu$talk. A declaration ($\mu$DECL) amounts to allocating memory for and initializing temporary variables to nil. A literal ($\mu$LIT) is simply evaluated and replaces the value of the last expression. An assignment ($\mu$ASS) updates the store at the variable’s location. Notice that the value stays around in case of another assignment, as in $v_1 := (v_2 := e)$.

When returning from a method ($\mu$LOCRET), the caller’s context is restored from the dump.

Message passing ($\mu$SEND), as mentioned, is the only control structure in $\mu$talk. First, the receiver is evaluated. The receiver’s class together with the selector uniquely determines the method to be invoked. If the class exists, it contains information about formal arguments and the method body. Then, the pseudo-variables self and super are bound to the receiver. After arguments have been bound to the actual parameters, the caller’s context is pushed on the dump. Notice that if the method is not found, findMethod will fail and the rule is not applicable.

The (special) case of sending a message to super ($\mu$SUPER) is similar to message passing except that no dynamic method lookup is required and the receiver is still self.

The construct new $c$ creates a new instance of class $c$ ($\mu$NEW). The instance is given a unique number through the function gensym and its instance variables are allocated and set to nil.

A plethora of primitive operations exists in a standard Smalltalk system. Here, we demonstrate the semantics for primitive 1, SmallInteger addition. The primitive returns if it succeeds ($\mu$PRIM$_1^1$), otherwise
(µPRIM\textsubscript{P}) it continues the execution, as if the primitive had not been there. All primitive methods follow this success/failure-pattern which is common Smalltalk practice.

The code after a call to a primitive method handles the case if the primitive fails, e.g. if the result of the addition of two SmallIntegers doesn’t fit in 32 bits. For this case, + is sent to the superclass (Integer) which handles the case when the argument or result is bigger than SmallInteger. This situation is shown later in the example code of section 3.5.

Finally, when there are no more expressions to execute, the final configuration is reached (µEND). Recall that the parser inserts self :: ↑\textsubscript{loc} at each method’s exit point. Hence, this rule only applies when all top-level expressions have been executed.

3 Smalltalk with Blocks

We now extend µtalk by adding blocks. Due to space limitations, only the changes in the definition are shown.

3.1 Syntax

New constructs:

\[
\text{Expr} \rightarrow \ldots \mid ↑\text{nonloc} \\
\text{Lit} \rightarrow \ldots \mid "[" \text{VAR}^* "|" \text{PGM} "]"
\]

A non-local return, ↑\text{nonloc}, marks the return from inside a block. Blocks without return expressions returns the value of the last expression in the block, thus ↑\text{loc} is inserted by the parser at the end of every block.

3.2 Semantics

Blocks in Smalltalk are defined by their arguments and a program body, possibly with some temporaries. They correspond to a λ-expression in functional languages, and the name closure is used synonymously as the result of evaluating a block. Closures are first-class citizens and can be assigned to variables, returned from methods, and sent along with messages:

\[ ω \in \text{Value} = \ldots + \text{Closure} \]

Upon receiving the message value, possibly along with some arguments, the closure begins executing its body. As opposed to λ-expressions, blocks must be invoked with all of its arguments.
Recall that in Smalltalk, blocks have access to the variable environment of the method in which it was created. Thus, by returning a block from a method, the method’s context (stack frame) can be accessed after the method has returned. This corresponds to the so-called “funarg” problem in LISP.

Another problem concerns return expressions inside blocks (non-local returns). When executed, control does not necessarily return to the object that invoked the block. Rather, it returns to the object that called the method in which the block was defined.

The difference between a local and non-local return is that a local return returns to the context on top of the dump, while a non-local return may return to a context somewhere further down the dump. It is considered an error to reference a missing context since, in Smalltalk, one can not return more than once from a method.

To be able to perform a non-local return from inside a block, a closure must have a reference to the context of its creator’s caller. The reference comes in form of a unique label, \( \lambda \in \text{Label} = \mathbb{N} \).

\[
\text{Closure} = \text{AEnv} \times \text{TEnv} \times \text{Block} \times \text{Label}
\]

The state is extended to hold the label of the creator’s caller (\( \lambda_c \)).

\[
\text{State} = \ldots \times \text{Label}
\]

\( \lambda_c \) serves two purposes: (1) to remember where to return to, in case of a non-local return, and (2), to fill out a closure, if a block is encountered.

When a context is pushed on the dump, it is labelled with a unique label. The context must also save \( \lambda_c \), the label on the right:

\[
\text{Context} = \text{Label} \times \ldots \times \text{Label}
\]

### 3.3 Auxiliary Functions

Some auxiliary functions must be redefined to handle blocks and closures (fig. 3). When `evalLit` evaluates a block into a closure it stores, as a part of the closure, the label of the context the block returns to upon a non-local return.

### 3.4 Semantic Rules

The new set of rules is given in fig. 3. There is also a new initial state, with \( \lambda_c = \text{undef} \):

\[
\tau_0 = \{ \rho_{ae0}, \rho_{te0}, \rho_{ie0}, \text{undef}, \sigma_0, \varepsilon, \varepsilon, \lambda_c \}
\]

Declarations (DECL), primitive calls (PRIM\(^1\)S,F), instantiation (NEW), and assignment (ASS) look the same way, with the exception of the label \( \lambda_c \), which is just passed along.
The rule for literals (LIT) is also the same, but notice that `evalLit` can now evaluate to a closure.

A local return (LOCRET) amounts to restoring the topmost context as before, with the noticeable addition of also restoring \( \lambda_c \). A non-local return (NONLOCRET) is similar to a local return with the difference that it doesn’t necessarily pop the topmost context. Instead, the control of execution is handed to the object that invoked the method in which the block was lexically defined. This context is pointed out by \( \lambda_c \).

A regular message send (SEND) or a send to super (SUPER) acts as before with the addition that the context placed on the dump is given a new unique label, \( \lambda_n \), and that it saves \( \lambda_c \) from the current state on the dump. Then, \( \lambda_n \) becomes the new \( \lambda_c \), a reference to the context which we just put on the dump. Remember that if the invoked method contains a block (evaluating to a closure) it may need to return to this context.

The special message `value` (VALUE), when sent to a closure, causes possible formal arguments to be allocated and bound to actual parameters. The current context is then stored on the dump as in (SEND) and the block’s body is evaluated in a state where \( \lambda_c \), the context to return to in case of non-local return, is taken from the closure. Notice that the closure “remembers” the environment of the method in which it was defined, which includes the value of `self`.

The rules presented are still disjoint, as long the method `value` is not defined in `HomeContext`. This requirement is easily enforced by the parser.

### 3.5 Example

Consider the following methods, shown here in concrete syntax. This example is deliberately contrived to exercise as many rules as possible.

<table>
<thead>
<tr>
<th>SmallInteger:</th>
</tr>
</thead>
<tbody>
<tr>
<td>fac</td>
</tr>
<tr>
<td>self fac: 1 and: [:a</td>
</tr>
<tr>
<td>fac: n and: b</td>
</tr>
<tr>
<td>(self &lt;= 0) ifTrue: [b value: n].</td>
</tr>
<tr>
<td>(self - 1) fac: n * self and: b</td>
</tr>
<tr>
<td>+ x</td>
</tr>
<tr>
<td>&lt;primitive 1&gt; self x.</td>
</tr>
<tr>
<td>↑super + x</td>
</tr>
<tr>
<td>False:</td>
</tr>
<tr>
<td>ifTrue: tBlk</td>
</tr>
<tr>
<td>↑nil</td>
</tr>
<tr>
<td>True:</td>
</tr>
<tr>
<td>ifTrue: tBlk</td>
</tr>
<tr>
<td>tBlk value</td>
</tr>
</tbody>
</table>

In fig. 4, the configurations generated by \( P[3 \text{ fac} \tau_0 ] \) are shown. Some noteworthy comments are given below:
• In (1), the rule for SEND is used, causing a context labelled 0 to be pushed on the dump. This context will later be referenced by the non-local return ↑a.

• In (3), the block [:a | ↑a ] has been evaluated into a closure c₁ which remembers context 0 as its non-local escape path.

• The inside of the method <= is not shown. Its result, however, can been seen in (9) where FALSE appears in the value field.

• In (10–11) the implementation of ifTrue: for False is seen: it chooses not to evaluate its block argument.

• In (25), after the multiplication and subtraction, fac:and: is invoked recursively. The following configurations are similar to (2–25) and not shown.

• Ultimately, fac:and: is invoked for the last time in (71). This time, <= returns TRUE (77). Subsequently, the ifTrue: method in True evaluates its block argument, causing the components in closure c₃ to be restored.

• In (80), the value of 3! is stored in n and is ready to be sent to the block which in (81), through a non-local return, which will take us right back to context labelled 0.

• In (83), we’re back in the original environment with the computed value 6.

4 Non-Standard Semantics

The operational semantics of Smalltalk presented in this paper can form the basis of analysis of Smalltalk programs, using the theory of abstract interpretation [CC77].

By changing the semantic rules and domain to some abstract counterpart, a conservative approximation of the semantics of a program can be found. The abstraction depends on the properties of interest in the analysis, and must also be done following the basic requirements of the theory of abstract interpretation.

One promising application of this idea is presented in [GPMT95]. In the paper, we describe how type inference can be accomplished by changing the concrete domain of values to an abstract domain consisting of object types. An object type is a list of ⟨classname, env⟩ pairs where env recursively describes the type of each instance variable.

For each variable in the program, the object type describes the classes which variable may be an instance of. Of course, this type may be an over-estimation of the actual run-time behavior.
As an example of the “collecting semantics”, consider update and its abstract counterpart. In update, \( \sigma[\ell \mapsto \omega] \) overwrote some variable’s old value. In an abstract setting, we want to add the new value (the type) to the existing type set:

\[
\sigma[\ell \mapsto \{\omega\}] \cup \text{lookup}(v, \rho_{ae} \otimes \rho_{te}, \rho_{ie}, \sigma)
\]

The type information can be used to optimize the program, for instance by implementing messages with function calls if the type is a singleton, or to verify that messages are always understood, by making sure that every class in the object type understands the message.

5 Implementation

Both \( \mu \)talk and \( \mu \)talk with blocks have been implemented and tested in Prolog and Haskell. Source code can be acquired by e-mail from the authors.

6 Future Work

To keep the presentation short and succinct, we have refrained from defining all of Smalltalk’s functionality. In a full definition, the state would be extended to carry an extra enviroment for global variables, and a class table so that classes (and their meta-classes) can be created dynamically.

More interesting work remains to be done in the area of analysis and optimization of Smalltalk and Smalltalk-like programs, like RTT-programs, as described in sec. 4. We imagine that analysis and optimization algorithms from other languages, for instance closure analysis, can be brought over to the Smalltalk community.

As indicated, the semantics for \( \mu \)talk could serve as a starting point for developing semantics of other object oriented languages.

Acknowledgements

This work has been founded partly by the Swedish National Board for Industrial and Tecnical Development (NUTEK project 93-3180) and partly by internal research foundings at the University of Mälardalen.
References


Figure 1: Syntax, semantic domains, and auxiliary functions for \( \mu \text{talk} \).
\[
\begin{align*}
\text{NIL} &= \{0, \text{"UndefinedObject"}\} & \rho_{\text{true}}(\text{"nil"}) &= 0 & \sigma_0(0) &= \text{NIL} \\
\text{TRUE} &= \{1, \text{"True"}\} & \rho_{\text{true}}(\text{"true"}) &= 1 & \sigma_0(1) &= \text{TRUE} \\
\text{FALSE} &= \{2, \text{"False"}\} & \rho_{\text{true}}(\text{"false"}) &= 2 & \sigma_0(2) &= \text{FALSE} \\
\rho_{\text{true}}(\text{"self"}) &= 3 & \sigma_0(3) &= \text{NIL} \\
\rho_{\text{true}}(\text{"super"}) &= 3 & \tau_0 &= (\rho_{\text{true}}, \rho_{\text{true}}, \rho_{\text{true}}, \text{undef}, \sigma_0, 4, \varepsilon)
\end{align*}
\]

\[
\begin{align*}
\rho'_{\text{true}} &= \rho_{\text{true}}[v_1 \mapsto \text{free}] \ldots [v_n \mapsto \text{free} + n - 1], (n > 0) \\
\sigma' &= \sigma[\text{free} \mapsto \text{NIL}] \ldots [\text{free} + n - 1 \mapsto \text{NIL}]
\end{align*}
\]

\[
\begin{align*}
\mu_{\text{DECL}}: & \quad (|v_1| \ldots v_n e_i, (\rho_{\text{true}}, \rho_{\text{true}}, \rho_{\text{true}}, \omega, \sigma, \delta)) \Rightarrow (e, (\rho_{\text{true}}, \rho_{\text{true}}, \rho_{\text{true}}, \omega, \sigma', \text{free} + n, \delta)) \\
\omega &= \text{evalLit}(\ell, \rho_{\text{true}}, \rho_{\text{true}}, \sigma) \\
\mu_{\text{LIT}}: & \quad (v :: e, (\rho_{\text{true}}, \rho_{\text{true}}, \rho_{\text{true}}, \omega, \sigma, \delta)) \Rightarrow (e, (\rho_{\text{true}}, \rho_{\text{true}}, \rho_{\text{true}}, \omega, \sigma, \text{free}, \delta)) \\
\sigma' &= \text{update}(v, \omega, \rho_{\text{true}}, \rho_{\text{true}}, \sigma)
\end{align*}
\]

\[
\begin{align*}
\mu_{\text{ASS}}: & \quad (\text{assign } v :: e, (\rho_{\text{true}}, \rho_{\text{true}}, \rho_{\text{true}}, \omega, \delta)) \Rightarrow (e, (\rho_{\text{true}}, \rho_{\text{true}}, \rho_{\text{true}}, \omega, \sigma'), \text{free}, \delta) \\
\omega_{\text{self}} &= \text{evalLit}(\text{free}, \rho_{\text{true}}, \rho_{\text{true}}, \rho_{\text{true}}, \sigma) \\
C &= (|e, \text{methods} \rangle \in \text{classTable} \\
\langle s, \text{formal}_1, \ldots \text{formal}_m, p \rangle &= \text{finalMethod}(s, C), (m \geq 0) \\
\omega &= \text{evalLit}(\ell, \rho_{\text{true}}, \rho_{\text{true}}, \rho_{\text{true}}, \sigma), (j = 1 \ldots m) \\
\rho_{\text{true}} &= \rho_{\text{true}}[\text{"self"} \mapsto \text{free}] [\text{"super"} \mapsto \text{free}] [\text{formal}_1 \mapsto \text{free} + 1] \ldots [\text{formal}_m \mapsto \text{free} + m] \\
\sigma' &= \sigma[\text{free} \mapsto \omega_{\text{self}}] [\text{free} + 1 \mapsto \omega_1] \ldots [\text{free} + m \mapsto \omega_m] \\
\delta' &= (e, \rho_{\text{true}}, \rho_{\text{true}}) :: \delta
\end{align*}
\]

\[
\begin{align*}
\mu_{\text{SEND}}: & \quad (\text{send } \ell_1 \ldots \ell_m :: e, (\rho_{\text{true}}, \rho_{\text{true}}, \rho_{\text{true}}, \omega, \delta)) \Rightarrow (e, (\rho_{\text{true}}, \rho_{\text{true}}, \rho_{\text{true}}, \omega, \delta')) \\
\omega_{\text{self}} &= \text{evalLit}(\text{"self"}, \rho_{\text{true}}, \rho_{\text{true}}, \rho_{\text{true}}, \sigma) \\
C &= (|e, \text{methods} \rangle \in \text{classTable} \\
\langle s, \text{informal}_1, \ldots \text{informal}_m, p \rangle &= \text{informalMethod}(s, C), (m \geq 0) \\
\omega &= \text{evalLit}(\ell_j, \rho_{\text{true}}, \rho_{\text{true}}, \rho_{\text{true}}, \sigma), (j = 1 \ldots m) \\
\rho_{\text{true}} &= \rho_{\text{true}}[\text{"self"} \mapsto \text{free}] [\text{"super"} \mapsto \text{free}] [\text{informal}_1 \mapsto \text{free} + 1] \ldots [\text{informal}_m \mapsto \text{free} + m] \\
\sigma' &= \sigma[\text{free} \mapsto \omega_{\text{self}}] [\text{free} + 1 \mapsto \omega_1] \ldots [\text{free} + m \mapsto \omega_m] \\
\delta' &= (e, \rho_{\text{true}}, \rho_{\text{true}}) :: \delta
\end{align*}
\]

\[
\begin{align*}
\mu_{\text{SUPER}}: & \quad (\text{super } c s \ell_1 \ldots \ell_m :: e, (\rho_{\text{true}}, \rho_{\text{true}}, \rho_{\text{true}}, \omega, \delta)) \Rightarrow (e, (\rho_{\text{true}}, \rho_{\text{true}}, \rho_{\text{true}}, \omega, \delta')) \\
i &= \text{gensym}() \\
C &= (|e, \text{super}, \text{methods}, \text{instvars} \rangle \in \text{classTable} \\
\ell_1 \ldots \ell_k &= \text{allInstVars}(C), (k \geq 0) \\
\rho'_{\text{true}} &= \rho_{\text{true}}[(v_i, 1) \mapsto \text{free}] \ldots [(v_k, i) \mapsto \text{free} + k - 1] \\
\sigma' &= \sigma[\text{free} \mapsto \text{NIL}] \ldots [\text{free} + k - 1 \mapsto \text{NIL}]
\end{align*}
\]

\[
\begin{align*}
\mu_{\text{NEW}}: & \quad (\text{new } c :: e, (\rho_{\text{true}}, \rho_{\text{true}}, \rho_{\text{true}}, \sigma, \delta)) \Rightarrow (e, (\rho_{\text{true}}, \rho_{\text{true}}, \rho_{\text{true}}, \{1, c\}, \sigma, \text{free} + k, \delta)) \\
\omega_1 &= \text{evalLit}(\ell_1, \rho_{\text{true}}, \rho_{\text{true}}, \rho_{\text{true}}, \sigma) \\
\omega_2 &= \text{evalLit}(\ell_2, \rho_{\text{true}}, \rho_{\text{true}}, \rho_{\text{true}}, \sigma) \\
\text{classOf}(\omega_1) &= \text{"SmallInteger"} \land \text{classOf}(\omega_2) = \text{"SmallInteger"} \land \text{classOf}(\omega_1 + \omega_2) = \text{"SmallInteger"} \\
\omega &= \text{evalLit}(\ell_1, \rho_{\text{true}}, \rho_{\text{true}}, \rho_{\text{true}}, \omega, \sigma) \\
\omega &= \text{evalLit}(\ell_2, \rho_{\text{true}}, \rho_{\text{true}}, \rho_{\text{true}}, \omega) \\
\text{classOf}(\omega_1) &= \text{"SmallInteger"} \lor \text{classOf}(\omega_2) = \text{"SmallInteger"} \lor \text{classOf}(\omega_1 + \omega_2) = \text{"SmallInteger"}
\end{align*}
\]

\[
\begin{align*}
\mu_{\text{PRIM}^2_3}: & \quad (\text{prim } 1 \ell_1 \ell_2 :: e, (\rho_{\text{true}}, \rho_{\text{true}}, \rho_{\text{true}}, \omega, \sigma, \delta)) \Rightarrow (\ell_1 \ell_2, (\rho_{\text{true}}, \rho_{\text{true}}, \rho_{\text{true}}, \omega, \sigma, \text{free}, \delta)) \\
\mu_{\text{PRIM}^2_4}: & \quad (\text{prim } 1 \ell_1 \ell_2 :: e, (\rho_{\text{true}}, \rho_{\text{true}}, \rho_{\text{true}}, \omega, \sigma, \delta)) \Rightarrow (\ell_1 \ell_2, (\rho_{\text{true}}, \rho_{\text{true}}, \rho_{\text{true}}, \omega, \sigma, \text{free}, \delta))
\end{align*}
\]

Figure 2: Initial state and semantic rules for \(\mu_{\text{talk}}\).
\[\rho'_{\nu} = \rho_{\nu} \text{ if } \nu \vdash \text{free}\]...

\[\sigma' = \sigma [\text{free} \rightarrow \text{NIL}]\]...

\[\text{DECL:}\]

\[\{ v_1 \ldots v_n \, | \, (\rho_{\nu_1}, \rho_{\nu_2}, \ldots, \rho_{\nu_n}, \sigma, \text{free}, \delta, \lambda_{0}) \Rightarrow (v_1 \rightarrow \text{free} + n - 1)\} (\text{if } n > 0)\]

\[\text{LIT:}\]

\[\{ \ell : e, (\rho_{\nu_1}, \rho_{\nu_2}, \ldots, \rho_{\nu_n}, \omega, \sigma, \text{free}, \delta, \lambda_{0}) \Rightarrow (\ell, \rho_{\nu_1}, \rho_{\nu_2}, \ldots, \rho_{\nu_n}, \omega, \sigma, \text{free}, \delta, \lambda_{0})\}\]

\[\text{ASS:}\]

\[\{ \text{assign } v : e, (\rho_{\nu_1}, \rho_{\nu_2}, \ldots, \rho_{\nu_n}, \omega, \sigma, \text{free}, \delta, \lambda_{0}) \Rightarrow (\rho_{\nu_1}, \rho_{\nu_2}, \ldots, \rho_{\nu_n}, \omega, \sigma, \text{free}, \delta, \lambda_{0})\}\]

\[\text{LOCREAT:}\]

\[\{ \text{locreat } \omega \rightarrow (\omega, \rho_{\nu_1}, \rho_{\nu_2}, \ldots, \rho_{\nu_n}, \omega, \sigma, \text{free}, \delta, \lambda_{0}) \Rightarrow (\omega, \rho_{\nu_1}, \rho_{\nu_2}, \ldots, \rho_{\nu_n}, \omega, \sigma, \text{free}, \delta, \lambda_{0})\}\]

\[\text{NONLOCREAT:}\]

\[\{ \text{nonlocreat } \omega \rightarrow (\omega, \rho_{\nu_1}, \rho_{\nu_2}, \ldots, \rho_{\nu_n}, \omega, \sigma, \text{free}, \delta, \lambda_{0}) \Rightarrow (\rho_{\nu_1}, \rho_{\nu_2}, \ldots, \rho_{\nu_n}, \omega, \sigma, \text{free}, \delta, \lambda_{0})\}\]

\[\text{SEND:}\]

\[\{\text{send } \ell_{m, n} \, s \, \ell_1 \ldots \ell_m \, : e, (\rho_{\nu_1}, \rho_{\nu_2}, \ldots, \rho_{\nu_n}, \omega, \sigma, \text{free}, \delta, \lambda_{0}) \Rightarrow (p, (\rho'_{\nu_1}, \rho'_{\nu_2}, \ldots, \rho'_{\nu_n}, \omega, \sigma, \text{free}, \delta, \lambda_{0})\}\]

\[\text{SUPER:}\]

\[\{\text{super } s \, \ell_1 \ldots \ell_m \, : e, (\rho_{\nu_1}, \rho_{\nu_2}, \ldots, \rho_{\nu_n}, \omega, \sigma, \text{free}, \delta, \lambda_{0}) \Rightarrow (p, (\rho'_{\nu_1}, \rho'_{\nu_2}, \ldots, \rho'_{\nu_n}, \omega, \sigma, \text{free}, \delta, \lambda_{0})\}\]

\[\text{VALUE:}\]

\[\{\text{send } \ell_{m, n} \, \text{value } \ell_1 \ldots \ell_m \, : e, (\rho_{\nu_1}, \rho_{\nu_2}, \ldots, \rho_{\nu_n}, \omega, \sigma, \text{free}, \delta, \lambda_{0}) \Rightarrow (p, (\rho''_{\nu_1}, \rho''_{\nu_2}, \ldots, \rho''_{\nu_n}, \omega, \sigma, \text{free}, \delta, \lambda_{0})\}\]

\[\text{NEW:}\]

\[\{\text{new } c : e, (\rho_{\nu_1}, \rho_{\nu_2}, \ldots, \rho_{\nu_n}, \omega, \sigma, \text{free}, \delta, \lambda_{0}) \Rightarrow (c, (\rho_{\nu_1}, \rho_{\nu_2}, \ldots, \rho_{\nu_n}, \omega, \sigma, \text{free}, \delta, \lambda_{0})\}\]

\[\text{PRIM}_1:\]

\[\{\text{prim } \ell_1 \ell_2 \, : e, (\rho_{\nu_1}, \rho_{\nu_2}, \ldots, \rho_{\nu_n}, \omega, \sigma, \text{free}, \delta, \lambda_{0}) \Rightarrow (\{\ell_{1, m}, (\rho_{\nu_1}, \rho_{\nu_2}, \ldots, \rho_{\nu_n}, \omega, \sigma, \text{free}, \delta, \lambda_{0})\}\}\]

\[\text{PRIM}_2:\]

\[\{\text{prim } \ell_1 \ell_2 \, : e, (\rho_{\nu_1}, \rho_{\nu_2}, \ldots, \rho_{\nu_n}, \omega, \sigma, \text{free}, \delta, \lambda_{0}) \Rightarrow (\ell, (\rho_{\nu_1}, \rho_{\nu_2}, \ldots, \rho_{\nu_n}, \omega, \sigma, \text{free}, \delta, \lambda_{0})\}\]

\[\{\text{end } \ell_1 \ell_2 \, : e, (\rho_{\nu_1}, \rho_{\nu_2}, \ldots, \rho_{\nu_n}, \omega, \sigma, \text{free}, \delta, \lambda_{0}) \Rightarrow (e, (\rho_{\nu_1}, \rho_{\nu_2}, \ldots, \rho_{\nu_n}, \omega, \sigma, \text{free}, \delta, \lambda_{0})\}\]

\[\{\text{figure 3: auxiliary functions and semantic rules for } \mu \text{ talk with blocks.}\]
Figure 4: Configurations visited for the factorial example.