Semantics of simple, object-oriented programming language (soopl)

Maciek Makowski

May 6, 2002

1 Abstract syntax

$Program ::= (ClassMap, Initializer)$
$ClassMap ::= ClassDecl\,list$
$Initializer ::= Expression$
$ClassDecl ::= (Id, ClassBody)$
$Id ::= \mathrm{x}_1, \mathrm{x}_2, ...$
$ClassBody ::= (Superclass, ClassAttributes, ClassMethodMap, Attributes, MethodMap)$
$Superclass ::= Id$
$ClassAttributes ::= Attributes$
$ClassMethodMap ::= MethodMap$
$Attributes ::= Id\,list$
$MethodMap ::= MethodDecl\,list$
$MethodDecl ::= (Id, Arguments, MethodBody)$
$Arguments ::= Id\,list$
$MethodBody ::= Statement\,list$
$Statement ::= Assignment|Return|Evaluation|Execution$
$Assignment ::= (Id, Expression)$
$Return ::= Expression$
$Evaluation ::= Expression$
$Execution ::= Expression$
$Expression ::= Identifier|Constant|BlockDecl|Message|Creation$
$Identifier ::= Id$
$Constant ::= 0, 1, 2, ...$
$BlockDecl ::= Statement\,list$
$Message ::= (Target, Id, ActualArguments)$
$Creation ::= Id$
$Target ::= Expression$
$ActualArguments ::= Expression\,list$

2 Semantic domains

- Identifiers: $Id = \{x_1, x_2, ...\}$
- Locations: $Loc = \{\alpha_1, \alpha_2, ...\}$
- Environments: $Env = (Id \to DV_\perp)$
- States: $S = (Loc \to SV_\perp)$
- Primitive values: $Primitive = \mathbf{N}$
- Classes: $Class = Loc \times Env \times Env \times Id\,list \times Env$
    1. superclass location
    2. class attributes environment
    3. class methods environment
    4. list of object attribute names
    5. object methods environment
- Objects: $Object = Loc \times Env$
    1. class location
    2. object attributes environment
- Methods: $Method = [Loc \to [Env \to [CExp \to CSt_\perp]]]$
    1. object/class location for which the method is being called
    2. local environment
    3. continuation to which the method result is passed
    4. final transformation on states
- Blocks: $Block = Env \times Env \times Env \times (Env \to Env \to Env \to\ CSt \to CSt)$
    1. global environment
    2. attribute/variable environment
    3. method environment
    4. function, which receives these three environments and yields the statement continuation transformation
- Expression continuations: $CExp = [EV \to CSt_\perp]$
- Statement continuation: $CSt = [S \to S_\perp + \{\text{error}\}_\perp]$
- Storable values: $SV = Class + Object + Primitive + Loc + \{\text{empty}\}$
- Denotable values: $DV = Loc + CExp + Method \times Id\,list + \{\text{unbound}\}$
- Expressable values: $EV = Loc + Primitive$

3 Miscellaneous operations

- Obtaining the list's head: hd $: \alpha\, list \to \alpha$
- Obtaining the list's tail: tl $: \alpha\, list \to \alpha\, list$
- Concatenating two lists: append $: \alpha\, list \to \alpha\, list \to \alpha\, list$
- Checking if element is in list: inlist $: \alpha \to \alpha\, list \to Boolean$
- Memory allocation: alloc $: S \to Loc \times S$
- Creating the environment based on identifier list: makeenv $: Id\, list \to S \to Env \times S$
  makeenv $= \lambda idl \in Id\, list.\lambda s \in S.$
  if $idl = []$ then $(\lambda id \in Id.\text{unbound}, s)$
  else let $(l, s') =$ alloc s in
  let $(\rho', s'') =$ makeenv $(\text{tl}\ idl)\ s'$ in
  $(\rho'[l/\text{hd}\ idl], s'')$
- Narrowing the environment so that only identifiers bound in second environment are bound in the resulting environment: envcast $: Env \to Env \to Env$
  envcast $= \lambda\rho_1 \in Env.\lambda\rho_2 \in Env.$
  $\lambda id \in Id.$ if $\rho_2\, id = \text{unbound}$ then unbound
  else $\rho_1\, id$
- Narrowing the environment so that only identifiers that are in the identifier list are bound in the resulting environment: envidcast $: Env \to Id\, list \to Env$
  envidcast $= \lambda\rho \in Env.\lambda idl \in Id\, list.$
  $\lambda id \in Id.$ if inlist $id\, idl$ then $\rho\, id$
  else unbound
- Composition of two environments: envcomb $: Env \to Env \to Env$
  envcomb $= \lambda\rho_1 \in Env.\lambda\rho_2 \in Env.$
  $\lambda id \in Id.$ if $\rho_2\, id = \text{unbound}$ then $\rho_1\, id$
  else $\rho_2\, id$

4 Additional assumptions

In the initial environment identifier Root is bound to location α_r. In the initial state in location α_r there is a class: $(\alpha_r, \lambda id \in Id.\text{unbound}, \lambda id \in Id.\text{unbound}, [], \lambda id \in Id.\text{unbound})$. This is the root class of the whole inheritance tree.

5 Semantic functions

$\mathcal{M}Program : Program \to Env \to S \to Env \times S$
$\mathcal{M}ClassMap : ClassMap \to Env \to S \to Env \times S$
$\mathcal{M}Initializer : Initializer \to Env \to S \to S$
$\mathcal{M}ClassDecl : ClassDecl \to Env \to S \to Env \times S$

$\mathcal{M}ClassBody : ClassBody \to Env \to S \to Loc \times S$
$\mathcal{M}ClMethodMap : MethodMap \to Env \to Env \to Env \to S \to Env$
$\mathcal{M}ClMethodDecl : MethodDecl \to Env \to Env \to Env \to S \to Env$
$\mathcal{M}MethodMap : MethodMap \to Env \to Env \to Env \to Id\,list \to Env \to S \to Env$
$\mathcal{M}MethodDecl : MethodDecl \to Env \to Env \to Env \to Id\,list \to Env \to S \to Env$
$\mathcal{M}MethodBody : MethodBody \to Env \to Env \to Env \to CSt$
$\mathcal{M}Statement : Statement \to Env \to Env \to Env \to CSt \to CSt$
$\mathcal{M}StatementList : Statement\,list \to Env \to Env \to Env \to CSt \to CSt$
$\mathcal{M}Expression : Expression \to Env \to Env \to Env \to CExp \to CSt$
$\mathcal{M}Arguments : ActualArguments \to Env \to Env \to Env \to Env \to Id\,list \to S \to S$

6 Semantic equations

Program: process the class map and then evaluate the initializer in resulting environment and state.

$\mathcal{M}Program[\![mkProgram(clm, init)]\!] = \lambda\rho \in Env.\lambda s \in S.$
$\quad$ let $(\rho', s') = \mathcal{M}ClassMap[\![clm]\!]\,\rho\,s$ in $(\rho', \mathcal{M}Initializer[\![init]\!]\,\rho'\,s')$

Class map: process all the class declarations in the list, each one in environment/state resulting from previous declaration.

$\mathcal{M}ClassMap[\![mkClassMap([])]\!] = \lambda\rho \in Env.\lambda s \in S.(\rho, s)$
$\mathcal{M}ClassMap[\![mkClassMap(cdl)]\!] = \lambda\rho \in Env.\lambda s \in S.$
$\quad$ let $(\rho', s') = \mathcal{M}ClassDecl[\![\mathsf{hd}\,cdl]\!]\,\rho\,s$ in $\mathcal{M}ClassMap[\![mkClassMap(\mathsf{tl}\,cdl)]\!]\,\rho'\,s'$

Initializer: treat the initializer as evaluation of the expression where attribute and method environments are empty and the statement continuation is identity.

$\mathcal{M}Initializer[\![mkInitializer(exp)]\!] = \lambda\rho \in Env.\lambda s \in S.$
$\quad \mathcal{M}Statement[\![mkEvaluation(exp)]\!]\,\rho$
$\qquad\qquad (\lambda id \in Id.\text{unbound})\,(\lambda id \in Id.\text{unbound})\,(\lambda s \in S.s)\,s$

Class declaration: process the class body and bind the identifier to the location where the class is placed.

$\mathcal{M}ClassDecl[\![mkClassDecl(id, cb)]\!] = \lambda\rho \in Env.\lambda s \in S.$
$\quad$ let $(l, s') = \mathcal{M}ClassBody[\![cb]\!]\,\rho\,s$ in $(\rho[l/id], s')$

Class body: find the superclass for the class being processed and:
• create the class attribute environment and combine the superclass' attribute environment with it
• process the class method map

• append the object attribute identifier list to the superclass' object attribute identifier list
• process the object method map
after that, allocate memory for the class and put the class in the allocated location.

$$\mathcal{M}ClassBody[\![mkClassBody(id, catr, cmm, atr, mm)]\!] = \lambda\rho \in Env.\lambda s \in S.$$
$$\text{let } mkClass(p, \rho_{ca}, \rho_{cm}, al, \rho_m) = s\,(\rho\,id) \text{ in}$$
$$\text{let } (\rho'_{ca}, s') = \text{makeenv}\, catr\, s \text{ in}$$
$$\text{let } \rho''_{ca} = \text{envcomb}\, \rho_{ca}\, \rho'_{ca} \text{ in}$$
$$\text{let } \rho'_{cm} = \mathcal{M}ClMethodMap[\![cmm]\!]\, \rho\, \rho''_{ca}\, \rho_{cm}\, s'$$
$$al' = \text{append}\, atr\, al \text{ in}$$
$$\text{let } \rho'_m = \mathcal{M}MethodMap[\![mm]\!]\, \rho\, \rho''_{ca}\, \rho'_{cm}\, al'\, \rho_m\, s'$$
$$(l, s'') = \text{alloc}\, s' \text{ in}$$
$$(l, s''[mkClass(\rho\,id, \rho''_{ca}, \rho'_{cm}, al', \rho'_m)/l])$$

Class method map: process the class method declarations in the list, each one using class method environment resulting from previous declarations.

$$\mathcal{M}ClMethodMap[\![mkMethodMap([])]\!] = \lambda\rho \in Env.\lambda\rho_{ca} \in Env.\lambda\rho_{cm} \in Env.\lambda s \in S.\rho_{cm}$$
$$\mathcal{M}ClMethodMap[\![mkMethodMap(mdl)]\!] = \lambda\rho \in Env.\lambda\rho_{ca} \in Env.\lambda\rho_{cm} \in Env.\lambda s \in S.$$
$$\mathcal{M}ClMethodMap[\![mkMethodMap(\text{tl}\, mdl)]\!]\, \rho\, \rho_{ca}\, (\mathcal{M}ClMethodDecl[\![\text{hd}\, mdl]\!]\, \rho\, \rho_{ca}\, \rho_{cm}\, s)\, s$$

Class method declaration: construct the operator F that takes method m as an argument, processes method body combining narrowed class attribute environment with local method environment (containing method arguments), narrowing the class method environment and binding the id of declared method to m in this environment. Then compute the fixpoint of F and bind the id of declared method to this fixpoint. Note the narrowing, which ensures, that only the identifiers bound in the environments of the class in which the method is declared are accessible to the method.

$$\mathcal{M}ClMethodDecl[\![mkMethodDecl(id, args, mb)]\!] = \lambda\rho \in Env.\lambda\rho_{ca} \in Env.\lambda\rho_{cm} \in Env.$$
$$\lambda s \in S.$$
$$\text{let } F = \lambda m \in Method.\lambda t \in Loc.\lambda\rho_{loc} \in Env.\lambda k \in CExpr.\lambda s \in S.$$
$$\text{let } mkClass(p, \rho'_{ca}, \rho'_{cm}, al, \rho_m) = s\,t \text{ in}$$
$$\mathcal{M}MethodBody[\![mb]\!]\, \rho$$
$$(\text{envcomb } (\text{envcast } \rho'_{ca}\, \rho_{ca})\, \rho_{loc}[k/\text{return}][t/\text{self}])$$
$$(\text{envcast } \rho'_{cm}\, \rho_{cm})[m/id]\, s$$
$$\text{in } \rho_{cm}[(fix\, F, args)/id]$$

Object method map: process the object method declarations in the list, each one using object method environment resulting from previous declarations.

$$\mathcal{M}MethodMap[\![mkMethodMap([])]\!] = \lambda\rho \in Env.\lambda\rho_{ca} \in Env.\lambda\rho_{cm} \in Env.\lambda al \in Id\,list.$$
$$\lambda\rho_m \in Env.\lambda s \in S.\rho_m$$
$$\mathcal{M}MethodMap[\![mkMethodMap(mdl)]\!] = \lambda\rho \in Env.\lambda\rho_{ca} \in Env.\lambda\rho_{cm} \in Env.$$

$$\lambda al \in Id\,list.\lambda\rho_m \in Env.\lambda s \in S.$$
$$\mathcal{M}MethodMap[\![mkMethodMap(\mathsf{tl}\,mdl)]\!]\,\rho\,\rho_{ca}\,\rho_{cm}\,al$$
$$(\mathcal{M}MethodDecl[\![\mathsf{hd}\,mdl]\!]\,\rho\,\rho_{ca}\,\rho_{cm}\,al\,\rho_m\,s)\,s$$

Object method declaration: construct the operator F that takes method m as an argument, processes method body combining narrowed object attribute environment with local method environment (containing method arguments), narrowing the object method environment and binding the id of declared method to m in this environment. Then compute the fixpoint of F and bind the id of declared method to this fixpoint. Narrowing works the same way as with class method declaration.

$$\mathcal{M}MethodDecl[\![mkMethodDecl(id, args, mb)]\!] = \lambda\rho \in Env.\lambda\rho_{ca} \in Env.\lambda\rho_{cm} \in Env.$$
$$\lambda al \in Id\,list.\lambda\rho_m \in Env.\lambda s \in S.$$
$$\mathsf{let}\ F = \lambda m \in Method.\lambda t \in Loc.\lambda\rho_{loc} \in Env.\lambda k \in CExp.\lambda s \in S.$$
$$\mathsf{let}\ mkObject(c, \rho_a) = s\,t\ \mathsf{in}$$
$$\mathsf{let}\ mkClass(p, \rho'_{ca}, \rho'_{cm}, al', \rho'_m) = s\,c\ \mathsf{in}$$
$$\mathcal{M}MethodBody[\![mb]\!]\,\rho$$
$$(\mathsf{envcomb}\ (\mathsf{envidcast}\ \rho_a\,al)\,\rho_{loc}[k/\mathrm{return}][t/\mathrm{self}])$$
$$(\mathsf{envcast}\ \rho'_m\,\rho_m)[m/id]\,s$$
$$\mathsf{in}\ \rho_m[(fix\,F, args)/id]$$

Method body: process the statements in the list, each one with statement continuation resulting from *following* statements; at the end of the list put the continuation that takes the expression continuation bound to identifier return in the attribute environment and passes to it the location bound to self in this environment. Thanks to this, each method that doesn't explicitely return something, will return the class/object it was called for (like in Smalltalk).

$$\mathcal{M}MethodBody[\![mkMethodBody([])]\!] = \lambda\rho \in Env.\lambda\rho_a \in Env.\lambda\rho_m \in Env.$$
$$\lambda s \in S.(\rho_a\,\mathrm{return})\,(\rho_a\,\mathrm{self})\,s$$
$$\mathcal{M}MethodBody[\![mkMethodBody(stl)]\!] = \lambda\rho \in Env.\lambda\rho_a \in Env.\lambda\rho_m \in Env.$$
$$\mathcal{M}Statement[\![\mathsf{hd}\ stl]\!]\,\rho\,\rho_a\,\rho_m\,(\mathcal{M}MethodBody[\![mkMethodBody(\mathsf{tl}\ stl)]\!]\,\rho\,\rho_a\,\rho_m)$$

Assignment: evaluate the expression with expression continuation that puts given value under location bound to id in the attribute environment.

$$\mathcal{M}Statement[\![mkAssignment(id, exp)]\!] = \lambda\rho \in Env.\lambda\rho_a \in Env.\lambda\rho_m \in Env.\lambda k \in CSt.$$
$$\mathcal{M}Expression[\![exp]\!]\,\rho\,\rho_a\,\rho_m\,(\lambda v \in EV.\lambda s \in S.k\,s[v/\rho_a\,id])$$

Return: evaluate the expression with expression continuation that passes given value to the expression continuation bound to return in the attribute environment.

$$\mathcal{M}Statement[\![mkReturn(exp)]\!] = \lambda\rho \in Env.\lambda\rho_a \in Env.\lambda\rho_m \in Env.\lambda k \in CSt.$$
$$\mathcal{M}Expression[\![exp]\!]\,\rho\,\rho_a\,\rho_m\,(\lambda v \in EV.\lambda s \in S.\rho_a\,\mathrm{return}\,v\,s)$$

Evaluation: evaluate the expression with expression continuation that discards the given value (only the side-effects will count).

$$\mathcal{M}Statement[\![mkEvaluation(exp)]\!] = \lambda\rho \in Env.\lambda\rho_a \in Env.\lambda\rho_m \in Env.\lambda k \in CSt.$$
$$\mathcal{M}Expression[\![exp]\!]\,\rho\,\rho_a\,\rho_m\,(\lambda v \in EV.k)$$

Execution: evaluate the expression with expression continuation that receives a location of a block and then executes the block in saved environments.

$$\mathcal{M}Statement[\![mkExecution(exp)]\!] = \lambda\rho \in Env.\lambda\rho_a \in Env.\lambda\rho_m \in Env.\lambda k \in CSt.$$
$$\mathcal{M}Expression[\![exp]\!]\,\rho\,\rho_a\,\rho_m\,(\lambda v \in EV.\lambda s \in S.$$
$$\mathsf{let}\ mkBlock(\rho, \rho_a, \rho_m, bl) = s\,v\ \mathsf{in}$$
$$bl\,\rho\,\rho_a\,\rho_m\,k\,s)$$

Statememt list: process all the statements in the list, each with statement continuation resulting from *following* statements (very similar to method body).

$$\mathcal{M}StatementList[\![[]]\!] = \lambda\rho \in Env.\lambda\rho_a \in Env.\lambda\rho_m \in Env.\lambda k \in CSt.k$$
$$\mathcal{M}StatementList[\![stl]\!] = \lambda\rho \in Env.\lambda\rho_a \in Env.\lambda\rho_m \in Env.\lambda k \in CSt.$$
$$\mathcal{M}Statement[\![\mathsf{hd}\ stl]\!]\,\rho\,\rho_a\,\rho_m\,(\mathcal{M}StatementList[\![\mathsf{tl}\ stl]\!]\,\rho\,\rho_a\,\rho_m\,k)$$

Identifier: if given id is unbound in the attribute environment then get the value bound to this id in the global environment, otherwise get the value bound to it in the attribute environment; pass the value to given expression continuation.

$$\mathcal{M}Expression[\![mkIdentifier(id)]\!] = \lambda\rho \in Env.\lambda\rho_a \in Env.\lambda\rho_m \in Env.\lambda k \in CExp.$$
$$\lambda s \in S.\ \mathsf{if}\ \rho_a\,id = \mathrm{unbound}\ \mathsf{then}\ k\,(\rho\,id)\,s\ \mathsf{else}\ k\,(s\,(\rho_a\,id))\,s$$

Constant: pass the constant value to given expression continuation.

$$\mathcal{M}Expression[\![mkConstant(n)]\!] = \lambda\rho \in Env.\lambda\rho_a \in Env.\lambda\rho_m \in Env.\lambda k \in CExp.k\,n$$

Block declaration: allocate memory for the block, process the statement list that constitutes the block and put the result in allocated location together with current environments; pass the block location to given expression continuation.

$$\mathcal{M}Expression[\![mkBlockDecl(stl)]\!] = \lambda\rho \in Env.\lambda\rho_a \in Env.\lambda\rho_m \in Env.\lambda k \in CExp.$$
$$\lambda s \in S.\mathsf{let}\ (l, s') = \mathsf{alloc}\ s\ \mathsf{in}$$
$$k\,l\,s'[mkBlock(\rho, \rho_a, \rho_m, \lambda\rho \in Env.\lambda\rho_a \in Env.\lambda\rho_m \in Env.\lambda k \in CSt.$$
$$\mathcal{M}StatementList[\![stl]\!]\,\rho\,\rho_a\,\rho_m\,k)/l]$$

Message: first evaluate the *target* expression; then:
- determine if the target is object or class
- find the method using target's method environment

• create local variable environment from the method argument identifier list
• combine the target attribute environment with it
• process the arguments
finally, call the method passing target location, local environment, given expression continuation and the state resulting from processing method arguments.

$$\begin{array}{l}
\mathcal{M}Expression[\![mkMessage(exp, id, args)]\!] = \lambda\rho \in Env.\lambda\rho_a \in Env.\lambda\rho_m \in Env. \\
\quad \lambda k \in CExp. \\
\quad \mathcal{M}Expression[\![exp]\!]\,\rho\,\rho_a\,\rho_m\,(\lambda t \in Loc.\lambda s \in S. \\
\qquad \mathsf{if}\ s\,t \in Object\ \mathsf{then} \\
\qquad\quad \mathsf{let}\ mkObject(c, \rho'_a) = s\,t\ \mathsf{in} \\
\qquad\qquad \mathsf{let}\ mkClass(p, \rho_{ca}, \rho_{cm}, al, \rho'_m) = s\,c\ \mathsf{in} \\
\qquad\qquad\quad \mathsf{let}\ (m, ids) = \rho'_m\,id\ \mathsf{in} \\
\qquad\qquad\qquad \mathsf{let}\ (\rho_{loc}, s') = \mathsf{makeenv}\ ids\,s\ \mathsf{in} \\
\qquad\qquad\qquad\quad \mathsf{let}\ \rho_{loc} = \mathsf{envcomb}\ \rho'_a\,\rho_{loc}\ \mathsf{in} \\
\qquad\qquad\qquad\qquad \mathsf{let}\ s'' = \mathcal{M}Arguments[\![args]\!] \\
\qquad\qquad\qquad\qquad\qquad \rho\,\rho_a\,\rho_m\,\rho_{loc}\,ids\,s'\ \mathsf{in} \\
\qquad\qquad\qquad\qquad m\,t\,\rho_{loc}\,k\,s'' \\
\qquad \mathsf{else} \\
\qquad\quad \mathsf{let}\ mkClass(p, \rho_{ca}, \rho_{cm}, al, \rho'_m) = s\,t\ \mathsf{in} \\
\qquad\qquad \mathsf{let}\ (m, ids) = \rho_{cm}\,id\ \mathsf{in} \\
\qquad\qquad\quad \mathsf{let}\ (\rho_{loc}, s') = \mathsf{makeenv}\ ids\,s\ \mathsf{in} \\
\qquad\qquad\qquad \mathsf{let}\ \rho_{loc} = \mathsf{envcomb}\ \rho'_a\,\rho_{loc}\ \mathsf{in} \\
\qquad\qquad\qquad\quad \mathsf{let}\ s'' = \mathcal{M}Arguments[\![args]\!] \\
\qquad\qquad\qquad\qquad\qquad \rho\,\rho_a\,\rho_m\,\rho_{loc}\,ids\,s'\ \mathsf{in} \\
\qquad\qquad\qquad\quad m\,t\,\rho_{loc}\,k\,s'')
\end{array}$$

Creation: find the class bound to id, create the object attribute environment using the id list in the class, allocate memory for the object and put a new object into resulting location; pass the object location to given expression continuation.

$$\begin{array}{l}
\mathcal{M}Expression[\![mkCreation(id)]\!] = \lambda\rho \in Env.\lambda\rho_a \in Env.\lambda\rho_m \in Env.\lambda k \in CExp. \\
\quad \lambda s \in S.\mathsf{let}\ (l, s') = \mathsf{alloc}\ s\ \mathsf{in} \\
\qquad \mathsf{let}\ mkClass(p, \rho_{ca}, \rho_{cm}, al, \rho'_m) = s'\,(\rho\,id)\ \mathsf{in} \\
\qquad\quad \mathsf{let}\ (\rho_a, s'') = \mathsf{makeenv}\ al\,s'\ \mathsf{in} \\
\qquad\qquad k\,l\,s''[mkObject(\rho\,id, \rho_a)/l]
\end{array}$$

Arguments: evaluate each expression in the expression list with the expression continuation that puts given value under the location bound to the next identifier in the argument id list. If there are more expressions than argument identifiers, don't evaluate the surplus expressions.

$$\begin{array}{l}
\mathcal{M}Arguments[\![mkArguments([])]\!] = \lambda\rho \in Env.\lambda\rho_a \in Env.\lambda\rho_m \in Env.\lambda\rho_{loc} \in Env. \\
\quad \lambda ids \in Id\,list.\lambda s \in S.s \\
\mathcal{M}Arguments[\![mkArguments(expl)]\!] = \lambda\rho \in Env.\lambda\rho_a \in Env.\lambda\rho_m \in Env.\lambda\rho_{loc} \in Env. \\
\quad \lambda ids \in Id\,list.
\end{array}$$

$$
\begin{array}{l}
\mathsf{if}\ ids = [\,]\ \mathsf{then}\ \lambda s \in S.s \\
\mathsf{else\ let}\ k = \mathcal{M}Arguments[\![mkArguments(\mathsf{tl}\, expl)]\!]\,\rho\,\rho_a\,\rho_m\,\rho_{loc}\,(\mathsf{tl}\, ids)\ \mathsf{in} \\
\qquad \mathcal{M}Expression[\![\mathsf{hd}\, expl]\!]\,\rho\,\rho_a\,\rho_m\,(\lambda v \in EV.\lambda s \in S.k\, s[v/\rho_{loc}\,(\mathsf{hd}\, ids)])
\end{array}
$$