The Impementation of SKOOL$^-$ in K-Maude

Grigore Roşu Dorel Lucanu

July 10, 2010

1 Introduction

TBD: Include the description of the untyped SKOOL, denoted here by SKOOL$^-$, from [1, 2]. This should be a clear and intuitive description, which is formalized then in the next section.

2 Implementation

The main steps we have to follow are:

1. define the syntax; for SKOOL$^-$, this mainly consists of adding new OO constructs to that of SIMPLE$^-$;
2. define the semantics, which has several substeps:
   (a) define the structure of the configuration; for SKOOL$^-$, this consists of adding some new cells to that of SIMPLE$^-$;
   (b) define the initial configuration and the rules for preprocessing (usually these are defined together because the initial configuration is depending on the result of the preprocessing);
   (c) add or modify step by step the computation rules, which define the operational semantics, and test them.

2.1 Syntax

The syntax of SKOOL$^-$ is obtained from that of SIMPLE$^-$ by

- replacing the definition of functions by that of methods,
- adding the definitions for classes and for the OO specific expressions, and
- modifying the definition for *Stmts*:

Exp ::=
  ... (*the same as in* SIMPLE$^-$)
  | `new` *Id* `()`
  | `new` *Id* `(` *List*{*Exp*} `)` [strict(2)]
  | `this`

| *Exp* `.` *Id* `()` [strict(1)]
| *Exp* `.` *Id* `(` *List*{*Exp*} `)` [strict(1 3)]
| `super` *Id* `()`
| `super` *Id* `(` *List*{*Exp*} `)` [strict(2)]
| *Exp* `instanceOf` *Id* [strict(1)]

Stmts ::= *VarDecl* | *MethodDecl* | *ClassDecl* | *Stmt*
| *Stmts* *Stmts*

VarDecl ::= `var` *Var* `;`

MethodDecl ::= `method` *Id* `()` *Stmt*
| `method` *Id* `(` *List*{*Id*} `)` *Stmt*

ClassDecl ::= `class` *Id* `{` *Stmts* `}`
| `class` *Id* `extends` *Id* `{` *Stmts* `}` ...

Id ::= `object`

In the module SKOOL⁻-UNTYPED-DESUGARED-SYNTAX, one macro is modified (function → method) and another few new ones are added:

EQUATION: `method` *F* `()` *S* = `method` *F* `(` `(.).List{Id}` `)` *S*
EQUATION: *E* `()` = *E* `(` `(.).List{Id}` `)`
EQUATION: *O* `.` *F* `()` = *O* `.` *F* `(` `(.).List{Id}` `)`
EQUATION: `new` *C* `()` = `new` *C* `(` `(.).List{Id}` `)`
EQUATION: *O* `.` *F* `()` = *O* `.` *F* `(` `(.).List{Id}` `)`
EQUATION: `super` *F* `()` = `super` *F* `(` `(.).List{Id}` `)`
EQUATION: `class` *C* `{` *Ss* `}` = `class` *C* `extends object` `{` *Ss* `}`

Let `SKOOL-UNTYPED-SYNTAX` denote the the module including the syntax and let `SKOOL-UNTYPED-DESUGARED-SYNTAX` denote the module including the desugaring macros.

Testing The syntax can be tested at this level following the next steps:

1. write a module `SKOOL-UNTYPED-SEMANTICS` including only the modules `SKOOL-UNTYPED-DESUGARED-SYNTAX` and `K` (nothing else);
2. write a module `SKOOL-UNTYPED-PROGRAMS` which includes the module `SKOOL-UNTYPED-SYNTAX` and some SKOOL⁻ programs;
3. write a module `SKOOL-UNTYPED` which includes `SKOOL-UNTYPED-SEMANTICS` and `SKOOL-UNTYPED-PROGRAMS`;
4. compile the above modules with the command `kompile.pl skool-untyped`, assuming that the file containing the module `SKOOL-UNTYPED` is *skool-untyped.k*. If the compiler does not complain, then it means the syntax is correctly written.

2.2 Configuration

The configuration for SKOOL⁻ is obtained from that of SIMPLE⁻ by adding the following new cells:

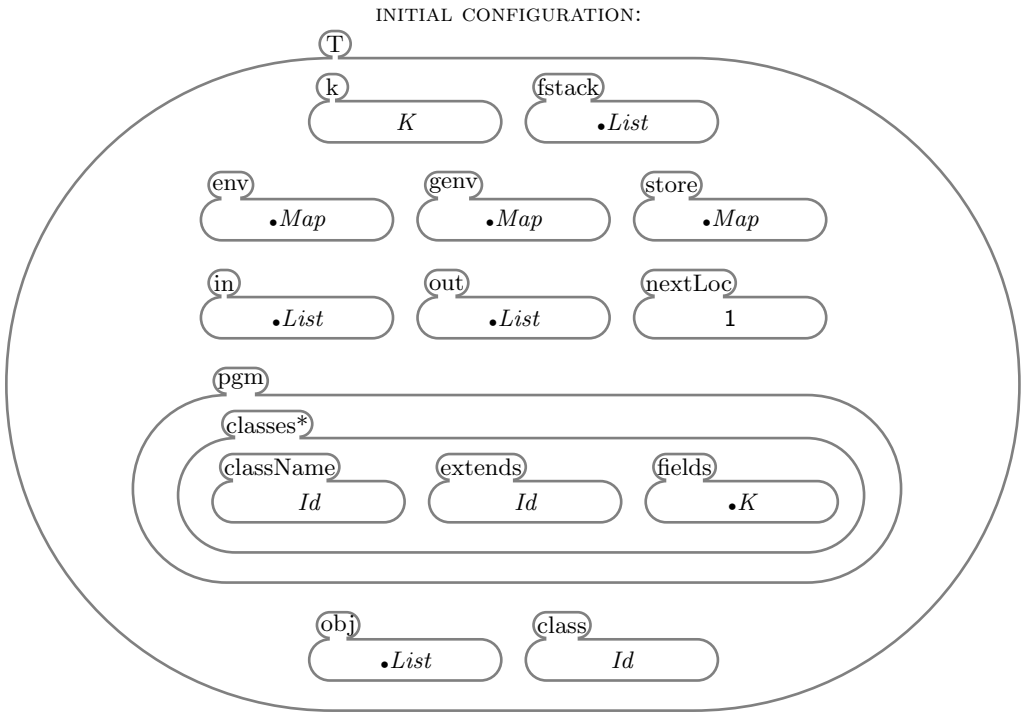


Figure 1: SKOOL$^-$ Configuration

- $\langle Id \rangle_{\mathsf{obj}}$, which includes the environment for the *this* object;
- $\langle Id \rangle_{\mathsf{class}}$, which includes the class name whose instance is supposed to be *this*;
- $\langle\langle\langle Id \rangle_{\mathsf{className}} \langle Id \rangle_{\mathsf{extends}} \langle \cdot K \rangle_{\mathsf{fields}} \rangle_{\mathsf{classes}*} \rangle_{\mathsf{pgm}}$, which includes the program structure as a bag of classes; the cell for a class includes a cell with its name, a cell with its superclass name, and a cell with the K-list of the fields. Keeping the fields as a K-list, allows to move them in the computation cell and to process them as any variable declaration.

The new configuration is represented in Fig. 1.

Testing At this stage we can bring all the modules defining the semantics of SIMPLE$^-$ and just replace in each module the name "SIMPLE" by "SKOOL". These modules replace the previous empty module for semantics. If we compile the file(s), then we get a syntax error at the rule processing functions. Because SKOOL$^-$ has no function, delete or comment this rule. Compile again, and we get a new error: in the initial configuration, the class cell includes a variable (*Id*). In the macro defining `run` we add the class cell with the initial content `object` (just to skip the error). Now it compiles fine. We may write some SIMPLE$^-$ programs without functions and execute them; it should work fine.

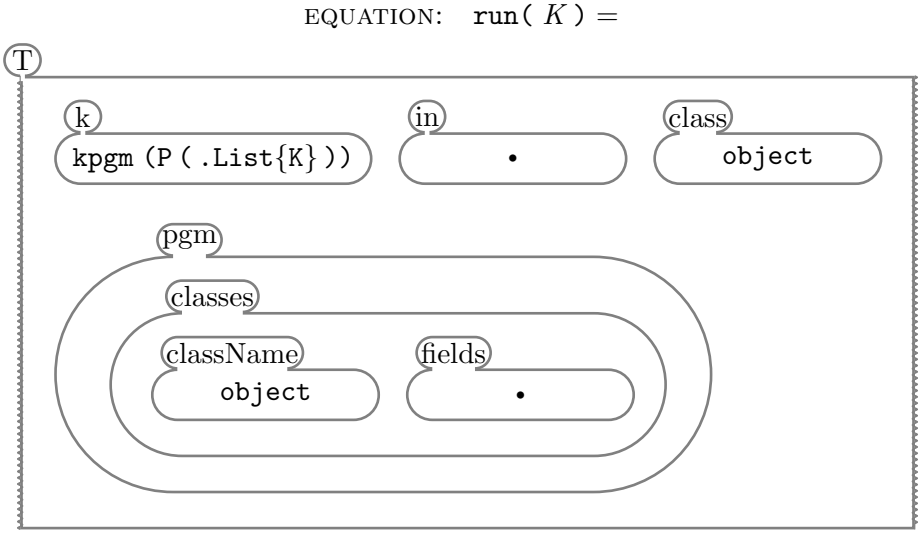


Figure 2: Initial configuration

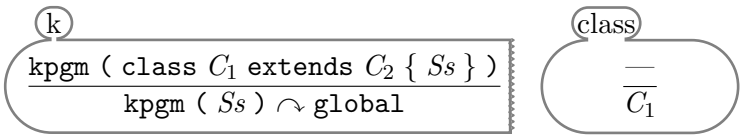
2.3 Preprocessing and the Initial Configuration

This step is different from that for SIMPLE$^-$ because the new language includes new syntactical constructs: classes, methods, and relationships between classes. The `pgm` cell is intended to store the structure of the program. The construction of the initial configuration for SKOOL$^-$ requires a preprocessing phase of the code in order to fill the `pgm` cell. This assumes the traversal of the program and to identify for each class its name, its immediate ancestor (parent), and its fields. The preprocessing of a SKOOL$^-$ program is different from that of SIMPLE$^-$. Recall that for SIMPLE$^-$, the program is seen as a K label and the rules preprocessing it could be concurrently applied; this is possible because the order in which the global variables and the functions are preprocessed does not matter. For SKOOL$^-$ we have to assign to each class its fields and the parent, therefore we have to sequentially preprocess the code. Our solution is to traverse the code by means of a K label `kpgm` and define the initial configuration as in Fig. 2. The program is represented by the K label `P`, which applied to the constant `.List{K}` produces a computation. If we write the preprocessing rules for `P ( .List{K} )`, as it is done for SIMPLE$^-$, then these rules are applied concurrently and could mess the classes structures. In order to avoid the concurrency, we wrap this computation using the K label `kpgm` and sequentialize translating it into $\curvearrowright$-list using the following rule:

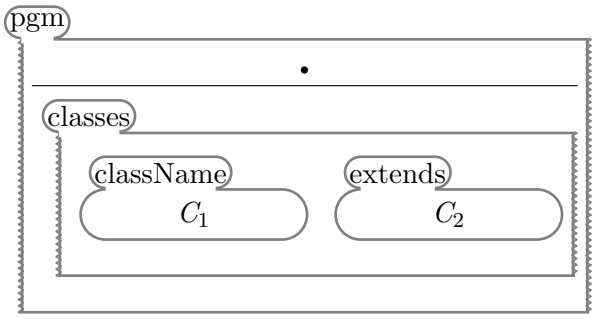
RULE: k: $\dfrac{\texttt{kpgm (} Ss_1 \;\; Ss_2 \texttt{)}}{\texttt{kpgm (} Ss_1 \texttt{)} \curvearrowright \texttt{kpgm (} Ss_2 \texttt{)}}$

The above rule is applied if and only if the head of the computation cell is a sequential composition. The elementary statements could be *global* or local to a class or method. A global statement is either a global variable declaration

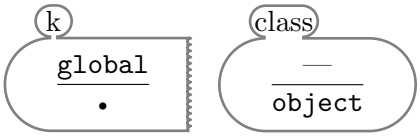
or a class declaration. The global variables are considered in SKOOL$^-$ as data members of the `object` class and therefore we can see the program as a sequence of classes declarations. A class is preprocessed with the following two rules:



RULE:

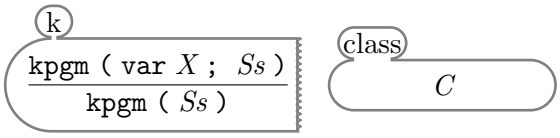


RULE:

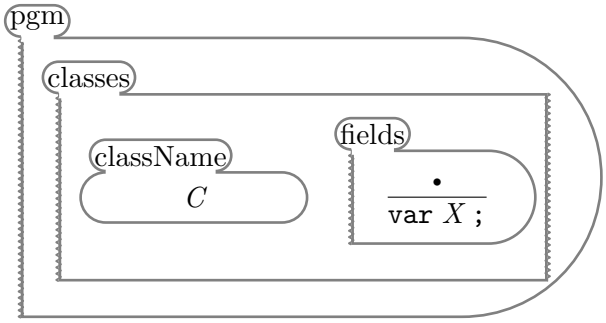


We use the class cell in order to process the statements local to a class; so, in the first above rule the statements Ss will be preprocessed as being local to the class C_1. The K label `global` is used to switch to the `object` class immediately after the preprocessing of a class is finished; in this way each variable declaration lying between two class declarations will be considered as a data member of the `object` class.

The processing of a variable declaration consists of adding it as a field the current class:



RULE:

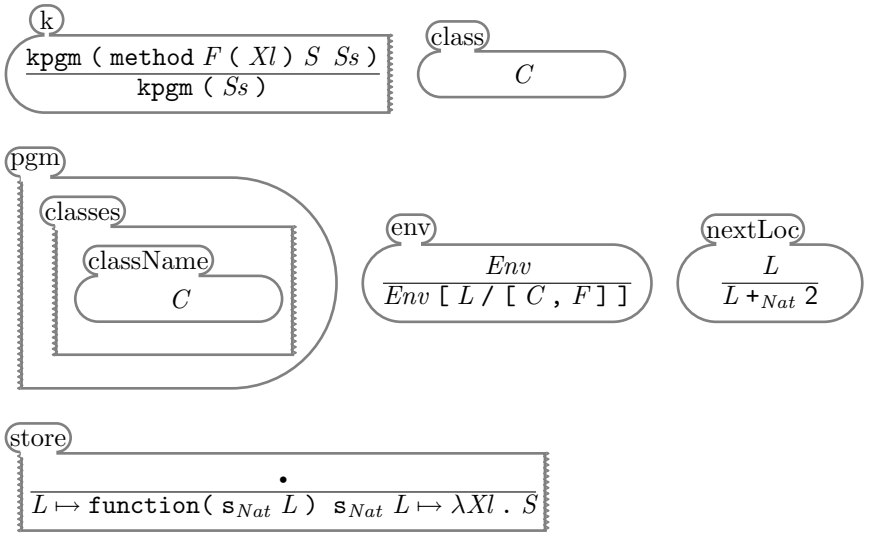


A similar rule is given for the case when the `kpgm` wraps only a single variable

declaration (this is the last declaration of a class or of the program). The array variables are processed in the same way.

For methods we followed the same line as that for SIMPLE$^-$ and we stored the methods as function values. In order to have unique names for these functions, we named them with pairs [class-name, method-name]:

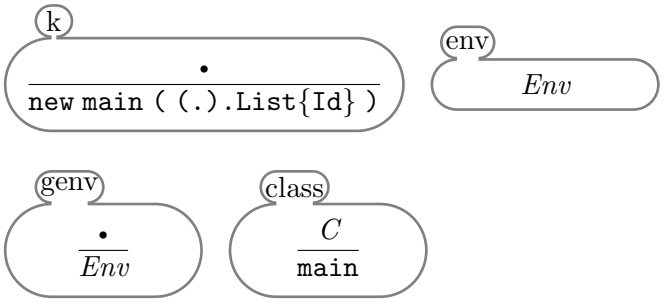
RULE:



A similar rule is given for the case when `kpgm` includes a unique method declaration (this is the last declaration in the class).

The preprocessing is finishing with the creation of the initial configuration for the computation (processing) phase:

RULE:



After all the program was preprocessed, the computation cell is empty. The computation phase starts with the construction of an object `main`. The environment, which includes the locations associated to the methods, is moved in the global environment cell.

Remark 1 *The above rules are replacing two rules (for functions and for loading* `main`*) in the module* `SKOOL-UNTYPED-PREPROCESSING`*.*

Testing In order to be sure that the preprocessing is correct, we need several SKOOL$^-$ programs, covering almost all cases: one ore more classes, zero, one or more data members, one or more methods, zero, one or more global variable.

Running these programs, in the final configuration the `k` cell should contain the expression for creating a new `main` instance (object) and the `pgm` cell should contain correct info (name, parent, and fields) for each class in the program.

2.4 Object values

Two things should be known about an object: 1) the relationship between data members and classes, and 2) the memory locations of the data members. The relationship between data members and classes is represented by an layered environment such that for each class C, lying on the path from the root to the current class in the hierarchy tree, there is a distinguished layer $C \mapsto E$, where E is the environment for the data members of C. This helps to instantiate the current object to each of these classes. An object value is represented by an operation `oenv` (O), which encapsulate the environment layer O, and a function `getLoc` (X, O), which returns the location of the name X from the layered environment O. The function `getLoc` (X, O) is defined by the following structural rules:

RULE: `getloc(` X `,` $C \mapsto$ `•` $Oenv$ `)` $\rightharpoonup$ `getloc(` X `,` $Oenv$ `)`

RULE: `getloc(` X `,` $C \mapsto$ `((` $Y \mapsto L$ `)` Env `)` $Oenv$ `)` $\rightharpoonup$
`if` $X ==_{Bool} Y$ `then` L `else getloc(` X `,` $C \mapsto Env$ $Oenv$ `) fi`

RULE: `getloc(` X `,` `•` `)` $\rightharpoonup$ `null`

where *null* represents "no location". Since *null* is not used in SIMPLE$^-$, where the natural numbers are used as locations, we used 0 as *null*:

Nat ::= `null`
| `getloc(` Id `,` $List$ `)` [strict]
RULE: `null` $\rightharpoonup$ 0

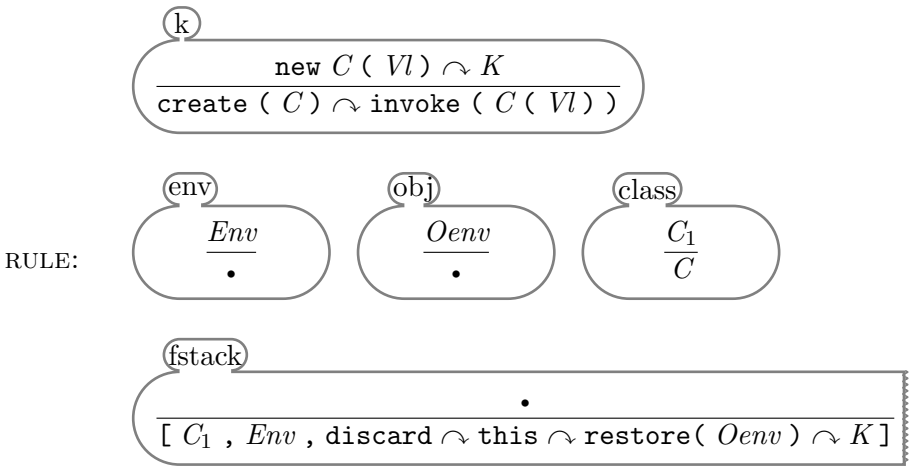
The start location is set on 1 (see Fig. 1).

Remark 2 *The definitions for* `oenv` *and* `getloc` *are included in the module* `SKOOL-UNTYPED-CONFIG`.

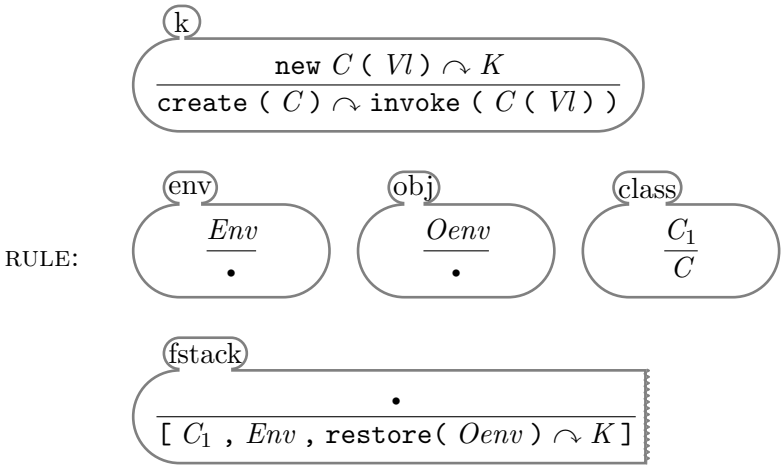
2.5 Expression Evaluation

2.5.1 New

The semantics of the `new` operator is given by two actions, memory allocation (create) and the constructor invocation, together with pushing in the stack the current environment, object environment, and the remaining computation. The value returned by constructor is discarded and the value of the new object is added in the head of the remaining computation. However, for the case of the initial object `main`, the value returned by `main` constructor should remain in the memory and therefore we consider a second rule handling this case.



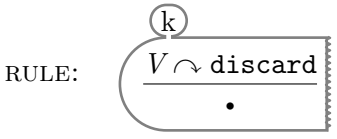
when C `=/=`$_{Bool}$ `main`



when K `==`$_{Bool}$ • `and`$_{Bool}$ C `==`$_{Bool}$ `main`

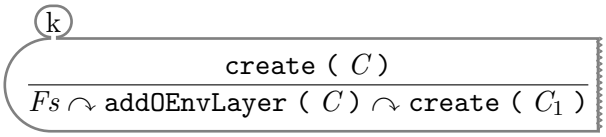
Remark 3 *Normally we should have some consistency between the current class (the content of the* $\langle\rangle_{\mathsf{class}}$ *cell) and the object* `this`*. The first above rule together with the one giving semantics to* `return` *breaks this consistency: the rule for* `return` *restores* C_1 *as the current class and* `this` *is an instance of* C*. The two classes* C *and* C_1 *could be totally different (on different paths in the hierarchy tree).*

The rules for *discard* and *restore* are very simple:

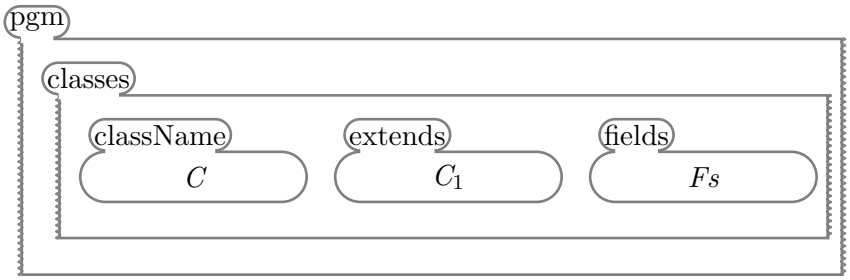


RULE:

The creation of an object consists in memory allocation for each data member and defining a layered environment in the `obj` cell:

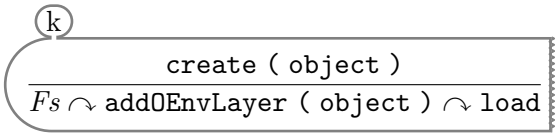


RULE:

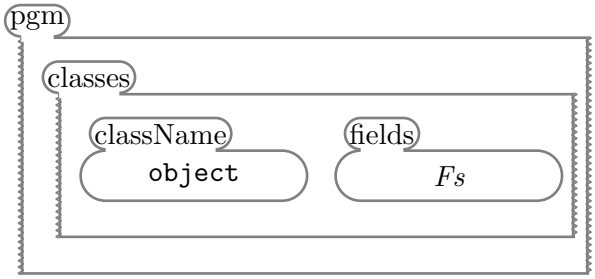


when C `=/=`$_{Bool}$ `object`

The last layer is that of the `object` class:

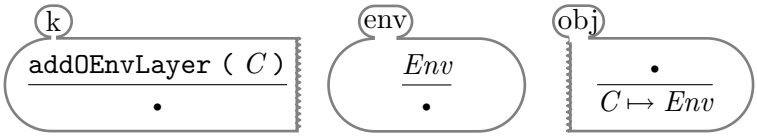


RULE:



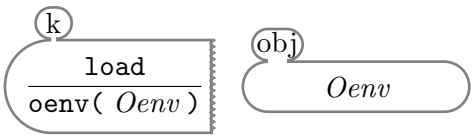
The label `addOEnvLayer` is processed by adding a new layer to the current object:

RULE:



The label `load` is processed by loading in the computation cell the environment of current object:

RULE:

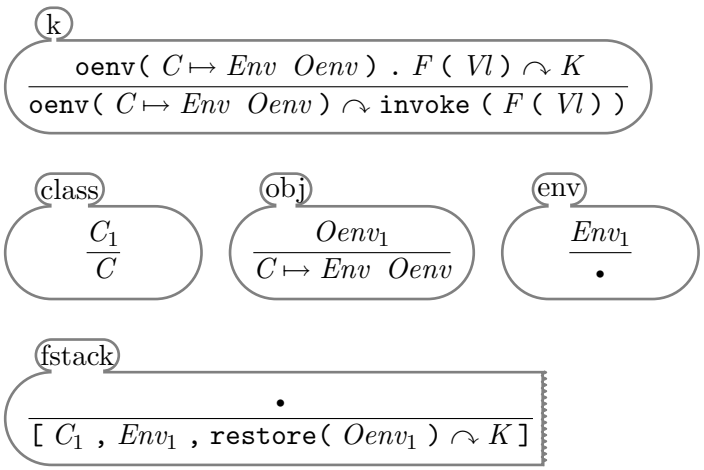


Testing Since the first computation is the evaluation of the expression `new main()`, we can test the semantics of the operator `new`. The execution of a program should terminate now with a configuration having in the head of the computation the new created object followed by the invocation of the constructor `main`. We can also check if the correctness of the layers of the new object.

2.5.2 Method Call

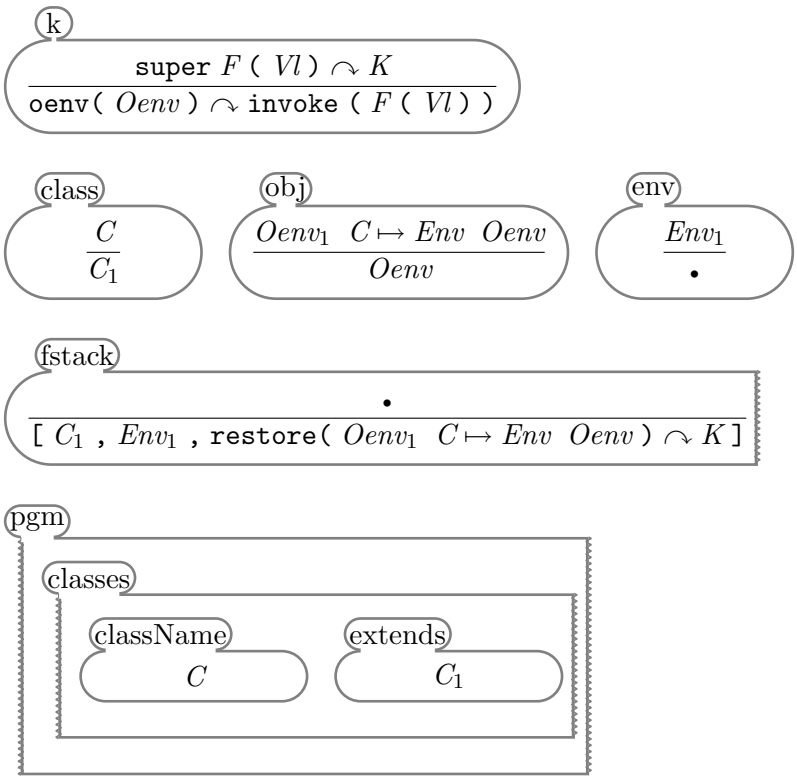
The semantics of a method call consists of pushing in the stack the current environments and the remaining computation, in order to prepare fresh ones for the method invocation. We have to cases: a method call from the current class

RULE:



or a method call from the superclass

RULE:



Testing This should be tested after the constructor invocation is tested for the `main` object.

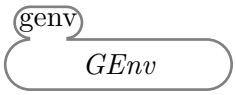
2.5.3 Method (Constructor) Invocation

This could refer both a constructor invocation, when a new object created, or an explicit method invocation. In the later case we have two subcases: either the method is a member of the current class or it is a member of an ancestor class. The semantics consists of two actions: first find the right function to be executed and then the execution of the function. This replaces the semantics of function calls from SIMPLE$^-$ (including the semantics of `return`). Recall that the name of an internal function (representing a method) is a pair [class-name, method-name].
The case when the method is in the current class:

k

$$\frac{\texttt{oenv(}\ C \mapsto Env\ \ Oenv\ \texttt{)} \curvearrowright \texttt{invoke}\ \texttt{(}\ F\ \texttt{(}\ Vl\ \texttt{)}\ \texttt{)}}{\texttt{oenv(}\ C \mapsto Env\ \ Oenv\ \texttt{)} \curvearrowright \texttt{function(}\ \texttt{s}_{Nat}\ L\ \texttt{)}\ \texttt{(}\ Vl\ \texttt{)}}$$

RULE:



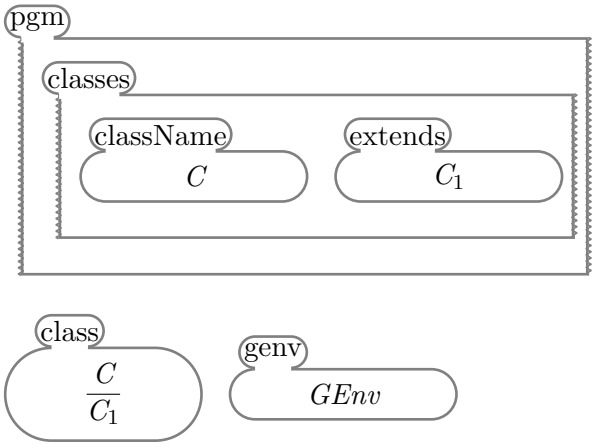
when $L := \texttt{getloc(}\ \texttt{[}\ C\ \texttt{,}\ F\ \texttt{]}\ \texttt{,}\ \texttt{@ny} \mapsto GEnv\ \texttt{)}$ $\texttt{and}_{Bool}$
$L\ \texttt{=/=}_{Bool}\ \texttt{null}$

The case when is looking in the parent class for the method:

k

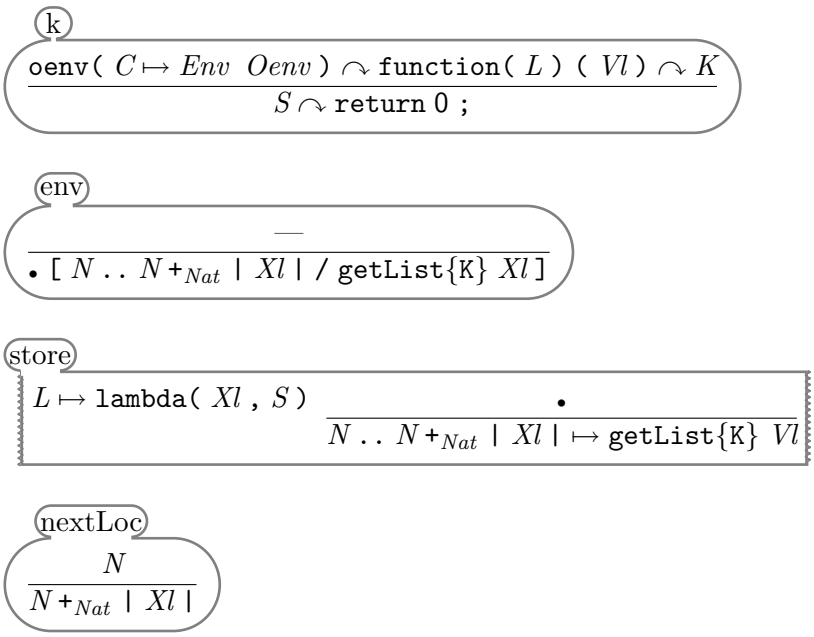
$$\frac{\texttt{oenv(}\ C \mapsto Env\ \ Oenv\ \texttt{)} \curvearrowright \texttt{invoke}\ \texttt{(}\ F\ \texttt{(}\ Vl\ \texttt{)}\ \texttt{)}}{\texttt{oenv(}\ Oenv\ \texttt{)} \curvearrowright \texttt{invoke}\ \texttt{(}\ F\ \texttt{(}\ Vl\ \texttt{)}\ \texttt{)}}$$

RULE:



when $\texttt{getloc(}\ \texttt{[}\ C\ \texttt{,}\ F\ \texttt{]}\ \texttt{,}\ \texttt{@ny} \mapsto GEnv\ \texttt{)}\ \texttt{==}_{Bool}\ \texttt{null}$

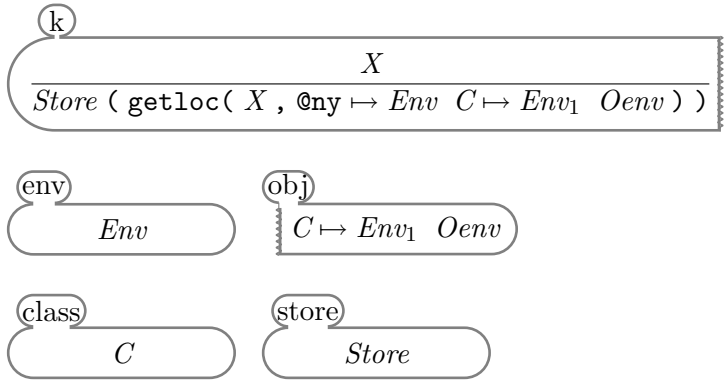
The load of the method's definition (as a function value):



Testing The final configuration reached at this stage should have in the head of the computation the first instruction from the `main` method which includes a variable name.

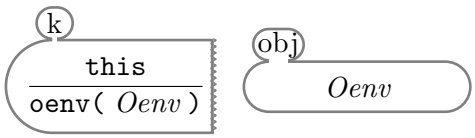
2.5.4 Variable Name Evaluation

In SKOOL$^-$, a variable could be a member of an object or local to a method. Therefore we have to look for its location in the environment of the abject, if it is a data member, or to the current environment, if it is local. We use `getloc` map, to find the location of the variable, and the map $S(L)$, which returns the value stored at location L in the store S (this is similar to a map $getValue(S, L)$):



A new rule for `this` is also added: this adds the instance of the current object environment, corresponding to the current class, as an encapsulated value using the *oenv* operation:

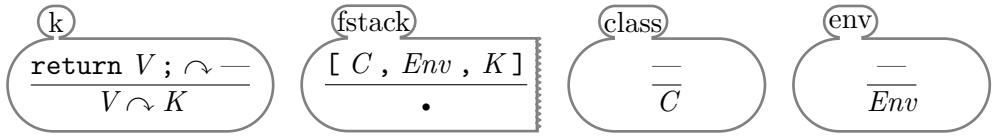
RULE:



2.5.5 Returning from a method call

The class, environment and computation from the top of the stack are restored; the returned value remains in the head of the computation:

RULE:

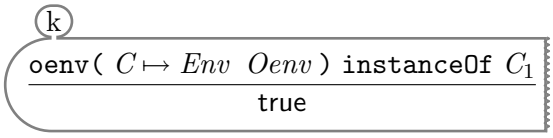


Testing At this point the implementation should be able to execute programs including hierarchies of classes, a class could having more than one methods, but without assignments.

2.6 InstanceOf

The semantics of `instanceOf` consists of examining the path from the current class to the root in the hierarchy tree until the object class is reached.
Instance of the current class:

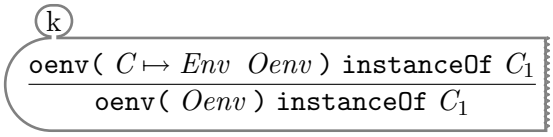
RULE:



when C `==`$_{Bool}$ C_1

Looking in the parent class:

RULE:

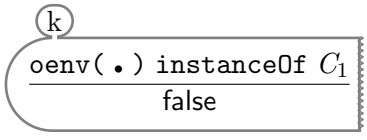


when

when C `=/=`$_{Bool}$ C_1

Not found:

RULE:



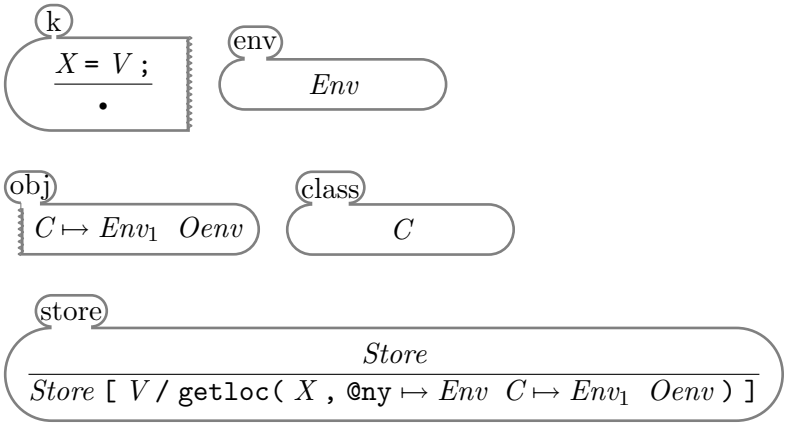
Testing We should test some special programs with not trivial hierarchies of classes and methods including the operator `instanceOf`, but, again, without assignments.

2.7 Instruction Processing

Only the semantics of the assignment is changed.

Assignment The rule for variable assignment must be changed because now the map *getloc* is used to find the location where the result value is stored.

RULE:



The rule for array component assignment remains unchanged because the location of the array is already computed when the assignment is executed.

Testing Here, the implementation should be able to execute any SKOOL$^-$ program.

3 Conclusion

Here is a comparison between SIMPLE$^-$ and SKOOL$^-$.

Syntax

1. `SKOOL-UNTYPED-SORTS` extends the smilar one from SIMPLE$^-$ by addind `List{Var}`.
2. `SKOOL-UNTYPED-EXPR` adds the OO specific expressions.
3. `SKOOL-UNTYPED-DECL` adds the declarations for methods and classes and removes the declarations for functions.
4. The instructions are the same.
5. `SKOOL-UNTYPED-STMTS`: the method and class declarations are added as statements and the function declarations are removed (as statements).

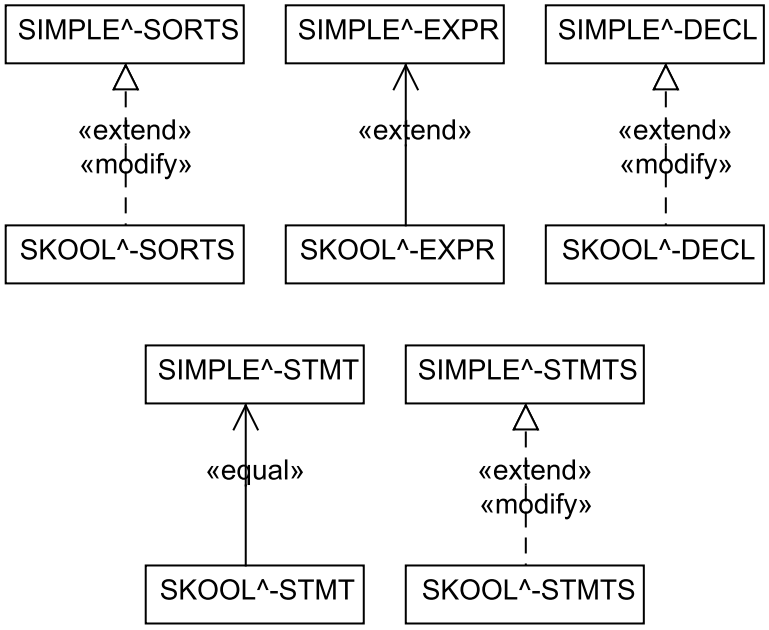


Figure 3: fig:Dependencies between the syntax modules implemented in SKOOL$^-$ resp. SIMPLE$^-$.

Semantics

- The `SKOOL-UNTYPED-CONFIG` module is obtained by extension; nothing removed or modified in the code inherited from `SIMPLE-UNTYPED-CONFIG`. It has been added new cells at the configuration, new values, new operations/labels.
- `SKOOL-UNTYPED-PREPROCESSING` is obtained by extension and modification; the rule processing functions was removed and the rules processing classes and methods were added.
- `SKOOL-UNTYPED-FUNCTION-CALL` it was completely modified. The rules for function calls replaced by the rules for method calls.
- `SKOOL-UNTYPED-INSTR-PROCESSING` is obtained by modification: only the rule for assignment was changed in the code inherited from the module `SIMPLE-UNTYPED-INSTR-PROCESSING`.

References

[1] G. Roşu. Defining Untyped SKOOL$^-$: a Simple Untyped Object-Oriented Language. CS422 - Programming Language Design (Fall 2007). `http://fsl.cs.uiuc.edu/images/8/8e/CS422-Fall-2007-13.pdf`

[2] G. Roşu. Elements of Object-Oriented Programming. CS422 - Programming Language Design (Spring 2010). `http://fsl.cs.uiuc.edu/images/4/4e/CS422-Spring-2010-elements-of-object-oriented-p`

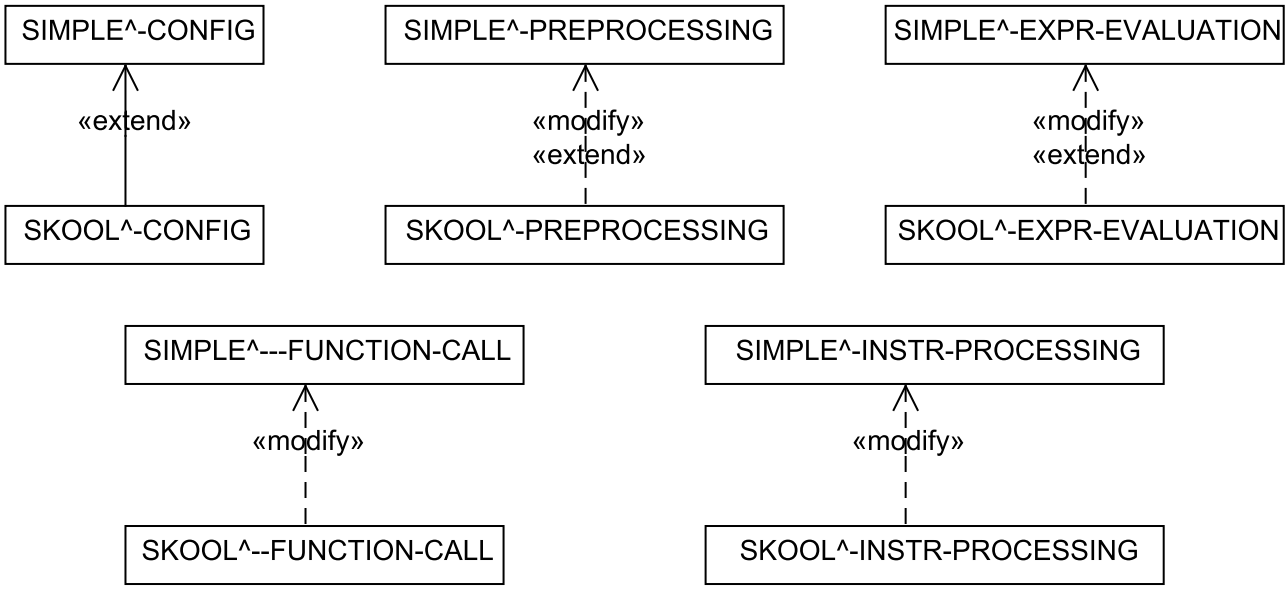


Figure 4: fig:Dependencies between the semantics modules implemented in SKOOL$^-$ resp. SIMPLE$^-$.