MP 7 – MiniJava Interpreter, Part 2

CS 421 – Spring 2013
Revision 1.0

Assigned Thursday, February 28, 2013
Due Tuesday, March 5, 2013, 9:30AM
Extension 48 hours (20% penalty)
Total points 50

1 Change Log

1.0 Initial Release.

2 Objective

You will continue writing an interpreter for MiniJava by translating the structural operational semantics (SOS) rules we have written for you into code. In this part, you will add objects. This will produce a fairly complete language, and will include most of the concepts needed to add arrays and more complex statements.

After completing this MP, you will have learned about the following concepts:

- interpret structural operational semantics of a programming language
- write an interpreter on an abstract syntax tree
- understand the role of the two-level store in implementing objects

This assignment is difficult. Please start early. Although the solution is structurally similar to MP6, quite a bit of it needs to change. Many of these changes are routine, but you will need to add or significantly modify about 50 lines of code. Since the types of almost all the functions changes, the MP7 skeleton includes new headers for all functions. We recommend you copy the function bodies from your MP6 and then make the required changes, rather than re-enter the code from scratch.

3 Style Requirements

In this MP, you will be expected to meet the following style requirements. Submissions that do not meet these requirements will not receive a grade until they are resubmitted with correct styling. (Note that these requirements will be checked manually by the grader and will not be enforced by `handin`, so acceptance by `handin` does not indicate correct style.)

- No long lines. Lines are 80 characters long.
- No tabs. Use spaces for indentation.
- Indents should be no more than 4 spaces, and must be used consistently.

4 Syntax and Semantics

We will continue to treat MiniJava as a dynamically-typed language, deciding the semantics of operators such as + at runtime. The new expressions we are implementing for MP7 are just two: `this` (which isn't even necessary — we will just treat `this` as a variable name) and `new C()`. Method call semantics change significantly, because the

receiver of the message now matters — it is used to find the class in which the method to be called is found. In addition, a variable reference can now be a field reference; the variable-lookup code checks if the variable is a local variable or argument, and if not, it checks if it is a field of the receiver. (We do not have the syntax $e.x$; only fields in the receiver can be accessed.)

The language is specified completely in the SOS rules in section 7. These say exactly how expressions should be evaluated and what statements should do. They may be confusing at first, but you will find that the interpreter is largely a direct transcription of those rules. SOS is your friend!

5 Testing and handin

- Download `mp7grader.tar.gz`. This tarball contains all the files you need: `minijavaast.ml` contains the abstract syntax, `minijavalex.mll` a lexer, and `minijavaparse.mly` a parser. In other words, you can transform programs to abstract syntax.
- As always, once you extract the tarball, you should rename `mp7-skeleton.ml` to `mp7.ml` and start modifying the file. You will modify only the `mp7.ml` file, and it is the only file that will be submitted.
- Compile your solution with `make`. Run the `./grader` to see how well you do; look in `tests` to see the test cases. As usual, it is a good idea to test your solution on more test cases; do this either by adding test cases to `tests` or using interactive execution.
- *Interactive use:* Functions to help with interactive debugging are defined in `testing.ml`. After running `make`, run OCaml by typing `ocaml`; then type `#use "testing.ml";;`. When you have finished some part of `eval`, you can test it by calling `evaluate` (only on variable-free expressions). Once you have finished enough of the other functions (e.g. `execStmt`, `evalMethodCall`), you can use the the functions `run` and `run_with_args` to test programs. Further specifics on interactive testing are included in `testing.ml`.
- Submit `mp7.ml` using the `handin` program.

6 Problems

The overall structure of the code for this assignment is just as in MP6: The main functions are `eval`, `applyOp`, and `execStmt`. These are defined precisely in the SOS rules in section 7. Almost all the rules are different from MP6, but in most cases only in simple ways. Specifically, the rules for expressions change to accommodate the use of the *two-level state*, and in particular the need to thread the store through expression evaluations. However, as for MP6, the evaluation of expressions involving strict operations is split into the recursive evaluation of the argument (done in `eval`) and the application of the operator (in `applyOp`); the definition of `applyOp` will not change.

As we did last week, we strongly recommend that you define and test the functions in the order given here.

The assignment is given in two parts: the implementation of two-level store and objects, and then the implementation of inheritance. The first is by far the more difficult; you will see that once you have objects implemented, implementing inheritance takes only a small amount of additional code. In the following, all the sections but the last relate to the first part of the assignment; inheritance is implemented in section 6.5. Similarly, the SOS rules are divided into those that apply to both cases, those that apply just to the non-inheritance case, and those that apply to the inheritance case. You can go right ahead and implement inheritance — all the test cases will run correctly — but we recommend two stages just for simplicity.

6.1 Two-level store

The purpose and operation of the two-level store were discussed in lectures 12, but you have seen it before, in classes like CS 232 and 241: it is the same idea as when the state is divided into a stack and a heap. We will use the terms "stack" and "environment" as synonyms, "environment" being the term traditionally used by language theorists while "stack" is used by compiler writers and architects; similarly, we use the terms "heap" and "store" as synonyms. The *state* is the combination of an environment and a store. In the SOS rules, we use ρ for environments, η for stores, and $\sigma = (\rho, \eta)$ for states.

So the basic idea ought to be somewhat familiar to you. Local variables and method parameters refer to locations in the stack; these may contain simple values (as in MP6), but may also contain pointers to objects in the heap. The heap, or store, contains objects; these may contain simple values and pointers to other objects in the heap. (In C and C++, pointers can also point into the stack, but Java does not allow this, and neither does MiniJava.)

A "pointer" is just a memory address. For our implementation, the store is a list of objects, and the address of each object is its position in the list. The store never shrinks — every object ever put in it remains there — so the location of any object doesn't change. When we create a new object — which occurs only when an expression of the form "new C()" is evaluated — we simply add it at the end of the store.

So, to summarize, we will have a first-level environment that maps local variables to values, just as in MP6, with the one difference that a variable's value can be a location. Thus:

```
type stackvalue = IntV of int | StringV of string
                | BoolV of bool | NullV | Location of location

and location = int

type environment = (varname * stackvalue) list
```

The heap, or store, contains objects. Each object gives its class, and a dictionary mapping field names to values. These are the same kinds of values that can go on the stack: simple values and pointers to other objects. Hence:

```
type heapvalue = Object of classname * environment

type store = heapvalue list
```

The *state* is just the combination of an environment and a store:

```
type state = environment * store
```

As in MP6, a new environment is created whenever a method call is made, and it lasts only as long as that method call. However — and this is the crucial point of this assignment — the store persists across calls. Whenever an expression is evaluated or a statement executed, it needs to be done in the *latest version* of the store. Environments come and go; the store abides.

Furthermore, unlike in MP6, *expression evaluation can change the store*. Specifically, evaluating "`new C()`" causes the store to expand (and, since an expression can contain subexpressions, which may themselves evaluate `new`, we have to assume that any expression can change the store). In MP6, expression evaluation could never in itself change the state; only statement execution could do that. In MP7, this is still true of the environment, but not of the store: if you evaluate an expression in state $\sigma = (\rho, \eta)$, then when you're done, ρ will be unchanged, but η may be changed.

The result of all this is that `eval` must "thread the store" — that is, every expression evaluation must return a new store, and that store must be passed along to subsequent expression evaluations. This shows up in the type of `eval`, which formerly returned just a `value`, but now returns a pair `stackvalue * store`.

There is one last important detail about the state, or rather, about environments. In a method call e`.f(`e_1`,...)`, e is evaluated first, and it must evaluate to a location. That location contains the "receiver" of the method. When executing `f`, any references to fields are resolved by looking in this object; the expression "`this`" returns that location as its value. So we will always assume that the environment contains a variable called `this`, whose value is a pointer to an object.

We now go into the detailed discussion of the functions for this assignment. The next few sections will change the interpreter to use the two-level state, and will implement objects and method calls correctly. In Section 6.5, we will add inheritance.

Figure 1 repeats the essential type definitions for the two-level store.

6.2 Utility functions

The functions in the first part of `mp7-skeleton.ml` are utilities to manipulate abstract syntax trees and states in simple ways.

```
type stackvalue = IntV of int | StringV of string
                | BoolV of bool | NullV | Location of location
and location = int

type environment = (varname * stackvalue) list

type heapvalue = Object of classname * environment

type store = heapvalue list

type state = environment * store

exception TypeError of string
exception RuntimeError of string
```

Figure 1: Essential declarations.

1. The following functions are unchanged from their MP6 versions, except possibly that the names of the types have changed:

   ```
   let rec asgn (id:id) (v:stackvalue) (env:environment) : environment =
   let rec binds (id:id) (env:environment) : bool =
   let rec fetch (id:id) (env:environment) : stackvalue =
   let rec mklist (i:int) (v:stackvalue) : stackvalue list =
   let rec zip (lis1:id list) (lis2:stackvalue list) : environment =
   let zipscalar (lis:id list) (v:stackvalue) : environment =
   let rec varnames (varlis:var_decl list) : id list =
   let getMethodInClass (id:id) (Class(_, _, _, methlis)) : method_decl =
   ```

   (Incidentally, `binds` is actually used in MP7.)

2. The next several functions deal specifically with the two-level state:

   (a) `extend` adds a new object to the store, by appending it at the end.

   ```
   let extend (st:store) (hval:heapvalue) : store =
     ...

   let sto1 = [Object("C", [("x", IntV 4); ("y", StringV "abc")])];;

   extend sto1  (Object("D", [("a", IntV 7); ("b", Location 0)]));;
   - : store = [Object("C", [("x", IntV 4); ("y", StringV "abc")]);
                Object("D", [("a", IntV 7); ("b", Location 0)])]
   ```

   (b) `storefetch` just gets the object at a given location in the store. Since the store is only extended when an object is created, and never shrinks, and since there is no way to obtain a location other than by allocating an object, we can safely assume that the location is valid.

   ```
   let storefetch (st:store) (loc:int) : heapvalue =
   ```

   (c) `asgn_fld` places a new value at a given field in an object. As with the environment itself, this doesn't actually change the object, but instead returns a new object that is like the old one but with the given field changed. (Note that you can use `asgn` here.)

   ```
   let asgn_fld (obj:heapvalue) (id:varname) (sv:stackvalue) : heapvalue =

   let obj1 = Object("C", [("x", IntV 4); ("y", StringV "abc")]);;

   asgn_fld obj1 "y" (Location 3);;
   - : heapvalue = Object ("C", [("x", IntV 4); ("y", Location 3)])
   ```

   (d) `asgn_sto` reassigns a location in the store, which is to say, it takes a given store and returns a store that is identical except at one location, where it has a new object. Again, we can assume the location of the assignment is valid.

```
let asgn_sto (sto:store) (loc:int) (obj:heapvalue) =

let sto2 = [Object("C", [("x", IntV 4); ("y", StringV "abc")]);
            Object("D", [("a", IntV 7); ("b", Location 0)])];;
let obj2 = Object ("C", [("x", IntV 4); ("y", Location 3)]);;

asgn_sto sto2 1 obj2;;

- : heapvalue list = [Object ("C", [("x", IntV 4); ("y", StringV "abc")]);
                      Object ("C", [("x", IntV 4); ("y", Location 3)])]
```

3. `getClass` finds a class with a given name in the program. This is a function we didn't need in MP6 because there was only one class. (Raise `TypeError` if the class is not found.)

   ```
   let getClass (c:id) (Program classlis) : class_decl =
   ```

4. `getMethod` looks for a method in a given class. We had this function in MP6, but since there was only one class, it didn't have the class name as an argument. (Raise `TypeError` if either the class is not found, or the method is not found in the class.)

   ```
   let getMethod (id:id) (c:id) (prog:program) : method_decl =
   ```

   We will revisit this function when we add inheritance (section 6.5).

5. `fields` gets a list of the names of the fields in a class.

   ```
   let fields (c:id) (prog:program) : string list =
   ```

6.3 Adding the two-level state to `eval`

This brings us to the heart of the matter: `applyOp` and `eval`.

In fact, `applyOp` still applies to values (after evaluation), and there are no operations on the new type of value (locations), so this does not change at all.

So, on to `eval`. As discussed above, the first problem is that we need to thread the store through calls to `eval`. The type of `eval` reflects this:

```
let rec eval (e:exp) ((env,sto) as sigma:state) (prog:program)
       : stackvalue * store
```

(The second argument uses an "as" pattern; this means it can be referred to as `sigma`, and at the same time its individual parts can be referred to as `env` and `sto`.)

As in MP6, you should follow the SOS rules closely; report type errors whenever there is a case that the SOS rules do not cover.

Our definition of `eval` is about 55 lines, as compared to about 30 in MP6. Of these 25 extra lines, five come from the clauses for the new expressions `new C` and `this`, and the other twenty from the increased complexity of the other clauses.

Starting with the clauses that were present in MP6, only variable references and method calls have substantive changes. For the others, the change is bookkeeping: making sure that the store is threaded through any recursive calls, and returned from this call. For variable reference, the big change is that a variable may refer to the field of an object. Look in the environment first; if the variable is not there, look in the object referred to by variable `this` and see if the variable is actually a field name of that object; if both these lookups fail, raise a type error.

The expression `this` is treated exactly as if it is a variable reference to a variable named `this`. As noted above (and see the next paragraph), you can always assume there is a variable named `this` in the environment.

Finally, we come to method calls, which have the form e`.f(`e_1`,...,`e_n`)`. Unlike MP6, we evaluate e; it must evaluate to a location ℓ, and the store will contain an object at that location, say `Object(C,...)`; *look for* `f` *in class* `C`. Then we proceed as in MP6: evaluate e_1, ..., e_n, bind their values to the formal parameters of `f`, and create an

environment by adding bindings for the local variables of `f`. We then do one more thing differently: place ℓ in the new environment as the value of `this`. (You can assume that there are no variables named `this`; if we wanted to go to the trouble, we could check and raise a type error if there were.)

The function `evallist` changes in the same way as most clauses of `eval`: it has to propagate store changes through each call to eval.

```
and evallist (el:exp list) ((env,sto) as sigma:state) (prog:program)
          : stackvalue list * store =
```

Function `evalMethodCall` can be copied verbatim from MP6.

6.4 Statements

The types of `execstmt` and `execstmtlis` do not change from MP6, but there is one substantive change in `execstmt`, which is in the clause for assignment. The issue is that assignment may be a change in the environment (as in MP6), or it may be a change in a field of the receiver object. As with variable evaluation above, look for a local variable first; if it is there, do the assignment as in MP6. Otherwise, look in the receiver object (the value of `this`); if this is an assignment to one of its fields, you need to create a new object with the changed value and then a new store with the new object; use `asgn_fld` and `asgn_sto`.

As you can see by comparing the SOS rules in section 7 with the rules in MP6, the definitions of the other statement types, and the definition of `execstmtlis`, are unaltered from MP6. You are now done with the first part of the assignment.

As with MP6, you can test the entire interpreter using the `run` and `run_with_args` functions; examples of using these are in `testing.ml`. You can add more test cases to the rubric by editing the `tests` file; just follow the pattern for the existing test cases.

6.5 Inheritance

The changes to allow Java-style inheritance are remarkably simple. In fact, they are confined to two auxiliary functions — `eval` does not change at all. Moreover, the types of the two function don't even change:

```
let fields (c:id) (prog:program) : string list =
let getMethod (id:id) (c:id) (prog:program) : method_decl =
```

`fields` is used to get the names of the fields of a new object, so that its environment can be formed; the difference is that it must now look recursively through all superclasses (stopping when a class has the empty string as its superclass name), and append all those field names together. Similarly, `getMethod` finds a method with a given name in a class; now it must look for it not only in the named class but in that class's superclasses. Both of these are straightforward recursive functions.

7 Formal specification

We provide a concrete syntax for this week's subset of MiniJava in Figure 2. We also repeat the definition of the abstract syntax in Figure 3.

The syntax for expression evaluation judgments is:

$$e, \sigma, \pi \Downarrow (v, \eta)$$

e is an `exp` (Figure 3), σ is a state (Figure 1), π a `program` (Figure 3), v a "stack value" (Figure 1), and η is a store. This asserts that e will evaluate to v in the given state and program (unless it raises some kind of exception), and that the evaluation may change the store to η. These items exactly match the arguments and result of `eval`, so that implementing these rules should be straightforward.

In practice, the rules often need to refer to the two parts of the state independently, so we often write the above judgment this way:

Program ::= *MainClassDecl ClassDecl**
MainClassDecl ::= `class Main {` *MethodDecl** `}`
ClassDecl ::= `class` *Id* `{` *FieldDecl** *MethodDecl** `}`
FieldDecl ::= `public`$^?$ *VarDecl*
MethodDecl ::= `public` *Type Id* `(` (*Type Id* (`,` *Type Id*)*)$^?$ `)` `{` *VarDecl** *Statement** `return` *Expression* `;` `}`
VarDecl ::= *Type Id* `;`
Type ::= `int` | `string` | `boolean` | *Id*
Statement ::= `{` *Statement** `}`
| `if (` *Expression* `)` *Statement* `else` *Statement*
| *Id* `=` *Expression* `;`
Expression ::= `(` *Expression* `)`
| *int* | *string* | `true` | `false` | *Id*
| `new` *Id* `()` | `this`
| *Expression* `.` *Id* `(` (*Expression* (`,` *Expression*)*)$^?$ `)`
| *Expression* (`+` | `-` | `*` | `/` | `==` | `<` | `&&` | `||`) *Expression*
| `!` *Expression*
Id ::= ⟨identifier⟩

Figure 2: MiniJava concrete syntax.

$$e, (\rho, \eta), \pi \Downarrow (v, \eta')$$

The judgments for statement execution have the same form as last week:

$$s, \sigma, \pi \Rightarrow \sigma'$$

where s is a statement, σ and σ' states, and π the program. This asserts that s, if executed in state σ, will change that state to σ' (whihc may include changes in both the environment and store). Again, this matches the arguments and result of `execStmt`.

The SOS rules themselves are given in four parts: the rules for expression evaluation, excluding `new` and method calls; those for statement execution; those for `new` and method calls without inheritance; and finally `new` and method calls with inheritance.

There are several notations used in the SOS rules:

$\rho(x)$ means `fetch` x σ
$\rho(x) \neq \perp$ means `binds` x σ = true
$\rho[v/x]$ means `asgn` x v σ
$\eta(n)$ means the value at location n in η
by abuse of notation, when v is the value `Location` n, $\eta(v)$ means the value at location n
$\pi(c)$ means the definition of class c in π.

```
type program = Program of (class_decl list)

and class_decl = Class of id * id
        * ((var_kind * var_decl) list)
        * (method_decl list)

and method_decl = Method of exp_type * id * (var_decl list)
        * (var_decl list) * (statement list) * exp

and var_decl = Var of exp_type * id

and var_kind = Static | NonStatic

and statement = Block of (statement list)
    | If of exp * statement * statement
    | Assignment of id * exp
      (* the following statement constructors not used this week *)
    | While of exp * statement | Println of exp
    | ArrayAssignment of id * exp * exp
    | Break | Continue

and exp = Operation of exp * binary_operation * exp
    | Integer of int
    | True
    | False
    | Id of id
    | Not of exp
    | Null
    | String of string
    | MethodCall of exp * id * (exp list)
    | This | NewId of id
      (* the following exp constructors not used this week *)
    | Array of exp * exp | Length of exp
    | NewArray of exp_type * exp | Float of float

and binary_operation = And | Or | LessThan | Plus | Minus
    | Multiplication | Division | Equal

and exp_type = ArrayType of exp_type | BoolType | IntType
    | ObjectType of id | StringType | FloatType

and id = string;;
```

Figure 3: MiniJava abstract syntax.

(INT) $i, (\rho, \eta), \pi \Downarrow (\mathsf{IntV}\ i, \eta)$

(STRING) $s, (\rho, \eta), \pi \Downarrow (\mathsf{StringV}\ s, \eta)$

(BOOL-TRUE) $\texttt{true}\ , (\rho, \eta), \pi \Downarrow (\mathsf{BoolV}\ \texttt{true}, \eta)$

(BOOL-FALSE) $\texttt{false}\ , (\rho, \eta), \pi \Downarrow (\mathsf{BoolV}\ \texttt{false}, \eta)$

(NULL) $\texttt{null}\ , (\rho, \eta), \pi \Downarrow (\mathsf{NullV}, \eta)$

(VAR) $x, (\rho, \eta), \pi \Downarrow (\rho(x),\ \eta)$ if $\rho(x) \neq \bot$

(FIELD) $x, (\rho, \eta), \pi \Downarrow (\eta(\rho(\texttt{this}))_2(x),\ \eta)$ if $\rho(x) = \bot\ \wedge\ \eta(\rho(\texttt{this}))_2(x) \neq \bot$

(THIS) $\texttt{this}\ , (\rho, \eta), \pi \Downarrow (\rho(this),\ \eta)$

(NOT) $\begin{array}{l} \texttt{!}\ e, (\rho, \eta), \pi \Downarrow (\mathsf{BoolV}\ \neg b, \eta') \\ \quad e, (\rho, \eta), \pi \Downarrow (\mathsf{BoolV}\ b, \eta') \end{array}$

(AND-TRUE) $\begin{array}{l} e_1 \texttt{\&\&}\ e_2, (\rho, \eta), \pi \Downarrow (v_2, \eta'') \\ \quad e_1, (\rho, \eta), \pi \Downarrow (\mathsf{BoolV}\ \texttt{true}, \eta') \\ \quad e_2, (\rho, \eta'), \pi \Downarrow (v_2, \eta'') \end{array}$

(AND-FALSE) $\begin{array}{l} e_1 \texttt{\&\&}\ e_2, (\rho, \eta), \pi \Downarrow (\mathsf{BoolV}\ \texttt{false}, \eta') \\ \quad e_1, (\rho, \eta), \pi \Downarrow (\mathsf{BoolV}\ \texttt{false}, \eta') \end{array}$

(OR-TRUE) $\begin{array}{l} e_1 \texttt{||}\ e_2, (\rho, \eta), \pi \Downarrow (\mathsf{BoolV}\ \texttt{true}, \eta') \\ \quad e_1, (\rho, \eta), \pi \Downarrow (\mathsf{BoolV}\ \texttt{true}, \eta') \end{array}$

(OR-FALSE) $\begin{array}{l} e_1 \texttt{||}\ e_2, (\rho, \eta), \pi \Downarrow (v_2, \eta'') \\ \quad e_1, (\rho, \eta), \pi \Downarrow (\mathsf{BoolV}\ \texttt{false}, \eta') \\ \quad e_2, (\rho, \eta'), \pi \Downarrow (v_2, \eta'') \end{array}$

(BINOP-INTEGER) $\begin{array}{l} e_1\ op\ e_2, (\rho, \eta), \pi \Downarrow (\mathsf{IntV}\ i_1\ op\ i_2, \eta'') \\ \quad e_1, (\rho, \eta), \pi \Downarrow (\mathsf{IntV}\ i_1, \eta') \\ \quad e_2, (\rho, \eta'), \pi \Downarrow (\mathsf{IntV}\ i_2, \eta'') \end{array}$ where $Op = +\ , -\ , or\ *$

(BINOP-INTEGER-DIV) $\begin{array}{l} e_1 / \ e_2, (\rho, \eta), \pi \Downarrow (\mathsf{IntV}\ i_1 \div i_2, \eta'') \\ \quad e_1, (\rho, \eta), \pi \Downarrow (\mathsf{IntV}\ i_1, \eta') \\ \quad e_2, (\rho, \eta'), \pi \Downarrow (\mathsf{IntV}\ i_2, \eta'') \end{array}$ if $i_2 \neq 0$

(BINOP-ZERO-DIV) $\begin{array}{l} e_1 / \ e_2, (\rho, \eta), \pi \Downarrow RuntimeError\ \texttt{"DivByZero"} \\ \quad e_1, (\rho, \eta), \pi \Downarrow (\mathsf{IntV}\ i_1, \eta') \\ \quad e_2, (\rho, \eta'), \pi \Downarrow (\mathsf{IntV}\ 0, \eta'') \end{array}$

(BINOP-LESS) $\begin{array}{l} e_1 / \ e_2, (\rho, \eta), \pi \Downarrow (\mathsf{BoolV}\ i_1 < i_2, \eta'') \\ \quad e_1, (\rho, \eta), \pi \Downarrow (\mathsf{IntV}\ i_1, \eta') \\ \quad e_2, (\rho, \eta'), \pi \Downarrow (\mathsf{IntV}\ i_2, \eta'') \end{array}$

Figure 4: SOS rules for evaluation, part 1

(STRING-PLUS1) $e_1 + e_2, (\rho, \eta), \pi \Downarrow (\mathsf{StringV}\ s_1 \wedge \text{to-string}\ v_2, \eta'')$
$e_1, (\rho, \eta), \pi \Downarrow (\mathsf{StringV}\ s_1, \eta')$
$e_2, (\rho, \eta'), \pi \Downarrow (v, \eta'')$

(STRING-PLUS2) $e_1 + e_2, (\rho, \eta), \pi \Downarrow (\mathsf{StringV}\ \text{to-string}\ v_1 \wedge s_2, \eta'')$
$e_1, (\rho, \eta), \pi \Downarrow (v_1, \eta')$
$e_2, (\rho, \eta'), \pi \Downarrow (\mathsf{StringV}\ s_2, \eta'')$

where to-string (StringV s) = s, to-string (IntV i) = string_of_int i, etc.

(EQUALS-STRING) $e_1 == e_2, (\rho, \eta), \pi \Downarrow (\mathsf{BoolV}\ s_1 = s_2, \eta'')$
$e_1, (\rho, \eta), \pi \Downarrow (\mathsf{StringV}\ s_1, \eta')$
$e_2, (\rho, \eta'), \pi \Downarrow (\mathsf{StringV}\ s_2, \eta'')$

(EQUALS-INT) $e_1 == e_2, (\rho, \eta), \pi \Downarrow (\mathsf{BoolV}\ i_1 = i_2, \eta'')$
$e_1, (\rho, \eta), \pi \Downarrow (\mathsf{IntV}\ i_1, \eta')$
$e_2, (\rho, \eta'), \pi \Downarrow (\mathsf{IntV}\ i_2, \eta'')$

(EQUALS-BOOL) $e_1 == e_2, (\rho, \eta), \pi \Downarrow (\mathsf{BoolV}\ b_1 = b_2, \eta'')$
$e_1, (\rho, \eta), \pi \Downarrow (\mathsf{BoolV}\ b_1, \eta')$
$e_2, (\rho, \eta'), \pi \Downarrow (\mathsf{BoolV}\ b_2, \eta'')$

(EQUALS-NULL) $e_1 == e_2, (\rho, \eta), \pi \Downarrow (\mathsf{BoolV}\ true, \eta'')$
$e_1, (\rho, \eta), \pi \Downarrow (\mathsf{NullV}, \eta')$
$e_2, (\rho, \eta'), \pi \Downarrow (\mathsf{NullV}, \eta'')$

(NOTEQUALS-NULL) $e_1 == e_2, (\rho, \eta), \pi \Downarrow (\mathsf{BoolV}\ false, \eta'')$ if $v \neq \mathsf{NullV}$
$e_2, (\rho, \eta), \pi \Downarrow (\mathsf{NullV}, \eta')$
$e_1, (\rho, \eta'), \pi \Downarrow (v, \eta'')$

(NOTEQUALS-NULL) $e_1 == e_2, (\rho, \eta), \pi \Downarrow (\mathsf{BoolV}\ false, \eta'')$ if $v \neq \mathsf{NullV}$
$e_2, (\rho, \eta), \pi \Downarrow (v, \eta')$
$e_1, (\rho, \eta'), \pi \Downarrow (\mathsf{NullV}, \eta'')$

Figure 5: SOS rules for evaluation (part 2)

(STMT-LIST) $s_1\texttt{;}\ \ldots\texttt{;}\ s_n\texttt{;}\ , \sigma_1, \pi \Rightarrow \sigma_{n+1}$
$s_1, \sigma_1, \pi \Rightarrow \sigma_2$
$\vdots$
$s_n, \sigma_n, \pi \Rightarrow \sigma_{n+1}$

(IF-TRUE) $\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2, (\rho, \eta), \pi \Rightarrow \sigma_1$
$e, (\rho, \eta), \pi \Downarrow (\mathsf{BoolV}\ true, \eta')$
$s_1, (\rho, \eta'), \pi \Rightarrow \sigma_1$

(IF-FALSE) $\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2, (\rho, \eta), \pi \Rightarrow \sigma_1$
$e, (\rho, \eta), \pi \Downarrow (\mathsf{BoolV}\ false, \eta')$
$s_2, (\rho, \eta'), \pi \Rightarrow \sigma_1$

(VAR-ASSIGN) $x = e, (\rho, \eta), \pi \Rightarrow (\rho[v/x], eta')$ if $\sigma(x) \neq \bot$
$e, (\rho, \eta), \pi \Downarrow (v, \eta')$

(FIELD-ASSIGN) $x = e, (\rho, \eta), \pi \Rightarrow (\rho, \eta'[\rho'[v/x]/\ell])$ if $\rho(x) = \bot \wedge \rho(\texttt{this}) = \text{Location}\ \ell$
$e, (\rho, \eta), \pi \Downarrow (v, \eta')$ $\wedge\ \eta(\ell) = \text{Object}(c, \rho') \wedge \rho'(x) \neq \bot$

Figure 6: SOS rules for statement execution

(NEW)	$\texttt{new}\, C, (\rho, \eta), \pi \Downarrow (\mathsf{Location}\ n, \eta')$	if	$\eta' = \eta@[\mathsf{Object}(C, [(x_1, \mathsf{NullV}); \ldots])]$ where $x_1, \ldots$ are the fields of class C and $n = \lvert\eta\rvert$
(METHOD-CALL)	$e_0\,.\, f(e_1, \ldots, e_n), (\rho, \eta), \pi \Downarrow (v, \hat{\eta})$ $e_0, (\rho, \eta), \pi \Downarrow (v_0, \eta')$ $e_1, (\rho, \eta'), \pi \Downarrow (v_1, \eta'')$ $\vdots$ $e_n, (\rho, \eta^n), \pi \Downarrow (v_n, \eta^{n+1})$ $s, (\rho', \eta^{n+1})], \pi \Rightarrow (\rho'', \eta^{n+2})$ $r, (\rho'', \eta^{n+2})', \pi \Downarrow (v, \hat{\eta})$	if	$v_0 = \mathsf{Location}\ \ell$ $\wedge\ \eta(\ell) = \mathsf{Object}(c, flds)$ $\wedge\ \pi(c)(f) = \quad \mathsf{Method}(\_,\mathsf{f},x_1 \ldots x_n,$ $y_1 \ldots y_m, s, r)$ $\wedge \rho' = ([(\mathsf{this}, v_0); (x_1, v_1); \ldots; (y_1, \mathsf{NullV}); \ldots]$

Figure 7: SOS rules for `this` and method calls, without inheritance

(NEW)	$\texttt{new}\, C, (\rho, \eta), \pi \Downarrow (\mathsf{Location}\ n, \eta')$	if	$\eta' = \eta@[\mathsf{Object}(C, [(x_1, \mathsf{NullV}); \ldots])]$ where $x_1, \ldots$ are the fields of class C and all its superclasses, and $n = \lvert\eta\rvert$
(METHOD-CALL)	$e_0\,.\, f(e_1, \ldots, e_n), (\rho, \eta), \pi \Downarrow (v, \hat{\eta})$ $e_0, (\rho, \eta), \pi \Downarrow (v_0, \eta')$ $e_1, (\rho, \eta'), \pi \Downarrow (v_1, \eta'')$ $\vdots$ $e_n, (\rho, \eta^n), \pi \Downarrow (v_n, \eta^{n+1})$ $s, (\rho', \eta^{n+1})], \pi \Rightarrow (\rho'', \eta^{n+2})$ $r, (\rho'', \eta^{n+2})', \pi \Downarrow (v, \hat{\eta})$	if	$v_0 = \mathsf{Location}\ \ell$ $\wedge\ \eta(\ell) = \mathsf{Object}(c, flds)$ $\wedge$ find $c\ f\ \pi = \quad \mathsf{Method}(\_,\mathsf{f},x_1 \ldots x_n,$ $y_1 \ldots y_m, s, r)$ $\wedge \rho' = ([(\mathsf{this}, v_0); (x_1, v_1); \ldots; (y_1, \mathsf{NullV}); \ldots]$ where find $c\ f\ \pi$ = definition of f in $\pi(c)$, if $\pi(c)(f) \neq \bot$, or find $s\ f\ \pi$, for s the superclass of c, otherwise

Figure 8: SOS rules for `this` and method calls, with inheritance