HW5 – Object-Oriented Programming

CS 476, Fall 2018
Due Oct. 25 at 2 PM

1 Instructions

Begin by downloading the file `hw5-base.ml` from the course website and renaming it to `hw5.ml`. This file contains the functions that you will use and modify in the homework. You will need to define new functions and modify `step_cmd`, but you should not need to modify any of the other predefined functions. Submit your completed `hw5.ml` via Gradescope. As always, please don't hesitate to ask for help on Piazza (`https://piazza.com/class/jkh8q52qrh06v`).

2 Evaluating Object-Oriented Programs

The file `hw5-base.ml` defines the types `exp` of expressions and `cmd` of commands for a simple Java-like language. It also defines two core functions: `eval_exp`, a big-step-style interpreter for expressions, and `step_cmd`, a small-step-style interpreter for commands. The following problems will ask you to extend the `step_cmd` function to handle the rest of the language.

You can test your code using the `run_prog` function, which takes a class table and a command and steps from the initial configuration until it gets stuck or terminates. The file includes a class table `ct1` corresponding to the following definitions:

```
class Shape extends Object{
  int id;

  int area(){ return 0; }
}

class Square extends Shape{
  int side;

  int area(){
    x = this.side;
    return x * x;
  }
}
```

To test the program `test0`, you would run

```
let (res_c, res_k, res_r, res_s) = run_prog ct1 test0;;
```

which should produce the output:

```
val res_c : cmd = Skip
val res_k : stack = []
val res_r : env = <fun>
val res_s : store = (<fun>, 1)
```

showing that `test0` has been evaluated and the resulting command, stack, environment, and store have been recorded as `res_c`, `res_k`, `res_r`, and `res_s` respectively. In this example, `res_c` should be `Skip`, `res_k` should be `[]` (the empty stack), `res_r` should be $\{\texttt{"s"} = p\}$ for some reference p, and `res_s` should be $\{p \mapsto \texttt{Obj("Square", [0; 3])}\}$, that is, it should map the reference p to an object of type `Square` with fields 0 (for the `id` field) and 3 (for the `side` field). To inspect the environment and store, you can look up `s` in the environment to find the associated reference, and then look up that reference in the store:

```
res_r "s";;
- : value option = Some (RefV 0)
store_lookup res_s 0;;
- : obj option = Some (Obj ("Square", [IntV 0; IntV 3]))
```

You should try running similar tests for the commands added in the problems. If your `step_cmd` function gets stuck on some input, `run_prog` will return the last configuration it successfully reached, so you can use its output to debug your code.

3 Problems

1. (6 points) Extend the provided `step_cmd` function with a case for `IfC`, the command-level if-then-else command. The condition of `IfC` should be an integer-valued expression that is treated as false if it evaluates to 0 and true otherwise, as described in the following rules:

$$\frac{(e, \rho, \sigma) \Downarrow 0}{(\texttt{IfC}\ \ (e, c_1, c_2), k, \rho, \sigma) \rightarrow (c_2, k, \rho, \sigma)}$$

$$\frac{(e, \rho, \sigma) \Downarrow v \qquad v \neq 0}{(\texttt{IfC}\ \ (e, c_1, c_2), k, \rho, \sigma) \rightarrow (c_1, k, \rho, \sigma)}$$

   Once you have completed this problem, `run_prog ct1 test1` should return a configuration in which the environment maps `x` to 2.

2. (5 points) Write a function `make_env : ident list -> value list -> env` that takes a list of identifiers `li` and a list of values `lv`, and returns an environment that maps each identifier in `li` to the corresponding element of `lv`. For instance, `make_env ["x", "y", "z"] [IntV 0, IntV 1, IntV 2]` should return an environment that maps `x` to 0, `y` to 1, and `z` to 2. You may want to use the predefined `update` function, which adds a single variable-value binding to an environment, and `empty_state`, the environment that has no bindings.

3. (14 points) The small-step semantics rules for method invocation and return are as follows:

$$\frac{\begin{array}{c}(e, \rho, \sigma) \Downarrow p \qquad \sigma(p) = \texttt{Obj}\ \ (c, ...) \qquad (\mathit{args}, \rho, \sigma) \Downarrow \mathit{vals} \\ \text{lookup\_method}(c, m) = \texttt{MDecl}\ \ (..., ..., \mathit{params}, \mathit{body})\end{array}}{(\texttt{Invoke}\ \ (x, e, m, \mathit{args}), k, \rho, \sigma) \rightarrow (\mathit{body}, (\rho, x) :: k, \{\texttt{this} = p, \mathit{params} = \mathit{vals}\}, \sigma)}$$

$$\frac{(e, \rho, \sigma) \Downarrow v}{(\texttt{Return}\ \ (e), (\rho_0, x) :: k, \rho, \sigma) \rightarrow (\texttt{Skip}, k, \rho_0[x \mapsto v], \sigma)}$$

These rules have been rephrased from the slides to more closely match the OCaml implementation, but have the same basic logic. Note that *params* and *vals* are lists of identifiers and values respectively; $\mathit{params} = \mathit{vals}$ means "map each element in *params* to the corresponding element of *vals*".

Extend `step_cmd` with cases for `Invoke` and `Return` according to these rules. The file already contains a function `store_lookup` for looking up values in σ, a function `eval_exps` for evaluating a list of expressions to a list of values, and a function `lookup_method` for looking up a method declaration in a class. You can use the `make_env` function from the previous problem to make the new environment $\{\texttt{this} = p, \mathit{params} = \mathit{vals}\}$.

Once you have extended `step_cmd` with both commands, `run_prog ct1 test2` should return a configuration in which the environment maps `x` to 0, and `run_prog ct1 test3` should return a configuration in which the environment maps `x` to 9.