Homework 4: Object Calculus

Due date: Thursday 11/3/2022 at 11:59pm

Overview

This homework consists of two parts:

1. Write some interesting programs in the object calculus.
2. Implement an interpreter for the object calculus.

Some important notes are as follows:

- The two components of the assignment are independent. We *strongly* encourage you to work through the first part to study for the object calculus component of the midterm.
- **You will need to use the myth cluster for Part 1. You can SSH onto them using your SUNet ID:** `ssh {your-sunet}@myth.stanford.edu`

  Part 2 was tested on the myth cluster (`myth.stanford.edu`), which has Python 3.8.10. If you have any issues with Python, you can use the cluster to do the this part of the assignment. You are free to develop your solution anywhere, but you should make sure that your solution works on the myth cluster, since that is where we will grade the assignment.

Part 1: Programming in the Object Calculus

In the first part of this assignment, you will write several programs in the object calculus. Object calculus programs are written in `*.objc` files. Object calculus program files contain a list of variable definitions or statements. The concrete syntax accepted by our object calculus parser is very similar to the syntax presented in lecture, except that we use `\o.o` to represent the method $\varsigma(o)\ o$. Refer to the test files in `test` and `problem.objc` for more examples of how to write object calculus programs.

Using the myth machines, you can evaluate object calculus files using `src/objc problem.objc`. Once you have completed all of the tasks, you can compare your output against the expected solutions using

```
src/objc problem.objc problem_test.objc --verify problem.objc.golden
```

Please do not edit the existing variable definitions in `problem.objc`, as these will be used by the autograder to evaluate your solutions. Add more tests or edit the tests for your answers in `problem_test.objc`. Do not add tests in `problem.objc` itself.

Your task is to write object calculus expressions that implement several functions on booleans and integers. Please see `problem.objc` for more detailed descriptions of the representation of these types within the object calculus.

Detailed instructions are as follows:

- **Write the implementation of the following three functions in** `problem.objc`**.** In the following specification of these functions, `n` and `m` denote arbitrary numerals encoding the integers n and m, and `b`, `b1` and `b2` refer to arbitrary encodings of booleans.
- `not`: Implement the `noter` object that peforms boolean negation of a boolean `b`.

- `and`: Implement the `ander` object that performs boolean and between booleans `b1` and `b2`.
- `add`: Implement the `adder` object that adds the numbers `n1` and `n2`.
- (Hint): Making clever use of method override is the key to implementing these functions.
- (Hint): For `add`, you might want to maintain additional state on the result of the `adder.eval`.
- (Note): It may take a few seconds to evaluate some of the tests in the `problem.objc`.

  The CAs' solution takes about 10 seconds to run `src/objc problem.objc problem_test.objc --verify problem.objc.golden`.

Part 2: Interpreter for the Object Calculus.

In the next part of this assignment, you will write an interpreter for the object calculus that can be used to evaluate object calculus expressions. In particular, you will implement the small-step operational semantics for the object calculus, which describe single steps of program evaluation. The operational semantics for the object calculus are described below:

$$\frac{e \mapsto e'}{e.l \mapsto e'.l} \text{ (Field-Access-Step)} \qquad \frac{o = [l_i = \varsigma(x_i)\ b_i;\ i \in 1 \ldots n]}{o.l_j \mapsto b_j\{x_j \coloneqq o\}} \text{ (Field-Access-Eval)}$$

$$\frac{e \mapsto e'}{e.l \Leftarrow \varsigma(x)\ b \mapsto e'.l \Leftarrow \varsigma(x)\ b} \text{ (Override-Step)}$$

$$\frac{o = [l_i = \varsigma(x_i)\ b_i,\ i \in 1 \ldots n]}{o.l_j \Leftarrow \varsigma(x)\ b \mapsto [l_j = \varsigma(x)\ b, l_i = \varsigma(x_i)\ b_i;\ i \in 1 \ldots n, i \neq j]} \text{ (Override-Eval)}$$

An object in the object calculus ($[\varsigma(x_i)\ b_i,\ i \in 1 \ldots n]$) is a concrete value that cannot take any steps. The "Step" rules in the semantics step the inner expression of a compound expression like field accessing or method override. Once the inner expression is a concrete object that cannot be stepped any further, the corresponding "Eval" rules actually manipulate the concrete object. The operational semantics described above utilize *substitution*, where a variable within an expression is replaced with a desired expression. The presence of variables and shadowing can make the implementation of substitution challenging. Therefore, we have provided a skeleton of an implementation and a strategy to implement your interpreter with substitution. In this assignment, we have provided rules and recursive derivations that attempt to align well to programming cases of recursive functions — the closer your implementation matches the definition of these functions, the more likely it is to be correct.

Task 1. In `interpreter.py`, implement the recursive function `free_vars`, a function that takes an expression and returns all free variables present in the expression. While implementing `free_vars`, you should implement `free_vars_method`, a helper function to find all free variables in a method $\varsigma(x)\ b$. The definition for the computation of free variables in the object calculus is provided below. The CAs' implementation is 12 lines of code.

$$\begin{aligned} FV(\varsigma(x)\ b) &= FV(b) - \{x\} \\ FV(x) &= \{x\} \\ FV([l_i = \varsigma(x_i)\ b_i;\ i \in 1 \ldots n]) &= \bigcup_{i=1}^{n} FV(\varsigma(x_i)\ b_i) \\ FV(o.l) &= FV(o) \\ FV(o.l \Leftarrow \varsigma(x)\ b) &= FV(o) \cup FV(\varsigma(x)\ b) \end{aligned}$$

Task 2. In `interpreter.py`, implement the recursive function `subst`, which substitutes the expression e_2 anywhere x appears within e_1. While implementing `subst`, you should implement `subst_method`, a helper

function that performs substitution on a method $\varsigma(x)\ b$. The definition of substitution provided below utilizes the calculation of free variables in an expression, and your implementation should as well. The notation $e_2\{x \coloneqq e_1\}$ can be read as "substitute e_2 for x in e_1". The CAs' implementation is 22 lines of code.

$$
\begin{aligned}
x\{x \coloneqq e\} &= e \\
y\{x \coloneqq e\} &= y, y \neq x \\
[l_i = \varsigma(x_i)\ b_i;\ i \in 1 \ldots n]\{x \coloneqq e\} &= [l_i = (\varsigma(x_i)\ b_i)\{x \coloneqq e\};\ i \in 1 \ldots n] \\
o.l\{x \coloneqq e\} &= (o\{x \coloneqq e\}).l \\
(o.l \Leftarrow \varsigma(y)\ b)\{x \coloneqq e\} &= (o\{x \coloneqq e\}).l \Leftarrow (\varsigma(y)\ b\{x \coloneqq e\}) \\
\varsigma(y)\ b\{x \coloneqq e\} &= \varsigma(y')\ ((b\{y \coloneqq y'\})\{x \coloneqq e\}), y' \notin FV(\varsigma(y)\ b) \cup FV(e) \cup \{x\}
\end{aligned}
$$

The final rule for substitution into a method is the trickiest, as we have to be careful about accidentally substituting into the wrong variable, or introducing incorrect variable bindings when substituting e into the scope of y. As discussed in lecture, we perform an operation called *alpha-renaming*, where we construct a new variable y' that does not appear in the free variables of the target method, the variable to replace, or in the expression to replace. Then, we substitute y' for y in the method before substituting e for x. This renaming ensures that the resulting method has a variable binder that is unique to the scope of the method. A simple way to generate a fresh variable name is to append a counter to the variable name and increment it until the variable no longer appears in the free variables (like `y0`, `y1`, ...).

Task 3. In `interpreter.py`, implement the recursive function `try_step`, which takes an expression e and returns `None` if e cannot take a step, or returns e' if $e \mapsto e'$. `try_step` should be implemented following the operational semantics of the object calculus described above at the beginning of Part 2. The CAs' implementation is 20 lines of code.

Detailed instructions:

- To implement these functions, refer to `src/objc.py` for the definitions of object calculus expressions.
- You can test your interpreter by running `python3 src/main_objc.py test/<test_name>`, which evaluates the contents of the input file. To check your interpreter against the expected output, run

  ```
  python3 src/main_objc.py test/<test_name> --verify test/<test_name>.golden
  ```

- You should also be able to run your interpreter on your solutions from Part 1 of the assignment using

  ```
  python3 src/main_objc.py problem.objc problem_test.objc --verify problem.objc.golden
  ```

  However, this may take a few seconds, as the evaluation strategy with explicit substitution is inefficient. The CAs' solution takes around 10 seconds to run the above command.

Submitting your work

- Please make sure you have edited `README.txt` to include your student ID number (the last 8-digit number).
- **Generate a tarball file `solution.tar.gz` by running `python3 src/submit.py` and upload the tarball file to Canvas.** Please make sure that the tarball file contains `interpreter.py`, `problem.objc`, and `README.txt`, and that the script gives you no errors or warnings.