CSE-321 Assignment 4
(100 points)

`gla@postech`

Due at 11:59pm, March 31

In this assignment, you will implement the operational semantics of the untyped λ-calculus.

The goal of this assignment is not just to implement what is called the operational semantics. Rather it is to develop the skill of *interpreting inference rules as algorithms,* which is crucial in implementing programming languages. That is, given a system of inference rules, we wish to extract an algorithm corresponding to the inference rules, and this assignment is designed to help you to develop such a skill.

Consider the three reduction rules for the reduction judgment $e \mapsto e'$ based on the call-by-value reduction strategy:

$$\frac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2}\ \mathit{Lam} \quad \frac{e_2 \mapsto e_2'}{(\lambda x.\, e)\ e_2 \mapsto (\lambda x.\, e)\ e_2'}\ \mathit{Arg} \quad \frac{}{(\lambda x.\, e)\ v \mapsto [v/x]e}\ \mathit{App}$$

Literally the rule *Lam* says that if e_1 reduces to e_1', then $e_1\ e_2$ reduces to $e_1'\ e_2$, and similarly for the other two rules. This literal interpretation answers the question of "why is a given *reduction* valid?" Conceptually the input to the problem is a reduction judgment $e \mapsto e'$, and the answer is either "yes" with a derivation tree that justifies the reduction, or "no" which means that the reduction is invalid. Thus the input to the problem is two expressions: an expression e and another expression e' which e may or may not reduce to.

In implementing the operational semantics, however, we would be interested in "how to reduce a given *expression*." In this case, the input to the problem is conceptually a certain expression e, and the answer is either another expression e' with a derivation tree of a judgment $e \mapsto e'$, or "no" which means that there is no expression that e reduces to. Therefore we need to interpret the reduction rules not literally but *algorithmically.* In this particular case, we need to interpret the reduction rules from the conclusion to the premises, *i.e.*, in the bottom-up way.

So let us interpret the reduction rules algorithmically. Since the input to the problem is an expression and the output is either another expression or "no," we introduce a function `step` with the following specification:

- `step` takes an expression e.
- `step` e returns another expression e' if e reduces to e'.
- `step` e raises an exception if there no expression that e reduces to.

Then how do we rewrite the rule *Lam*, for example, in terms of the function `step`? Intuitively the rule *Lam* should be interpreted as follows:

1. Consider the case in which `step` takes $e_1\ e_2$ as an argument.

2. `step` makes a recursive call to e_1 because it needs to determine the expression that e_1 evaluates to, if any.

3. If `step` e_1 returns e_1', we return $e_1'\ e_2$.

4. If `step` e_1 raises an exception, we propagate it (by not installing an exception handler).

The other two rules can be interpreted in a similar way. The goal of this part is to implement such a function `step` to implement the reduction rules.

In implementing the function `step`, you will perhaps have to extract functions from the inductive definitions of:

- $FV(e)$ for free variables in expression e,
- $[e'/x]e$ for substituting e' for x in e,
- $e \equiv_\alpha e'$ for the α-equivalence between e and e'.

Unlike the previous assignments in which we gave all the types of the functions to be implemented, this assignment does not provide the specification for these functions except for their inductive definitions, all of which can be found in the course notes. All we care about is the correctness of `step` and nothing else.

The reason why we do not give out the specification for these functions (other than their inductive definitions) is to teach students an important principle in software development: *design and specification*. Half the battle in software development is actually to figure out "what to implement" rather than "how to implement." For example, the implementation of `reach`, `distance`, and `weight` in Assignment 3 would have been a lot more difficult had students been instructed to start from scratch. The programming part in this assignment is essentially no different: you will spend most of your time *designing* your code rather than actually writing it.

So our advice is: *think a lot before you type anything on the screen.* You don't even have to turn on your computer before you finalize the design – what functions to implement, their types, their invariants, and so on. You might well be tempted to start with a (bad) design without giving enough consideration to its correctness, but eventually it will waste you more time than it saves. So again, think a lot before you type anything on the screen. The amount of time you will spend (or waste) doing this assignment will be directly proportional to the number of times you ignore this advice.

Programming instruction

Download `hw4.zip` from the course webpage or the handin directory, and unzip it on your working directory. It will create a bunch of files on the working directory.

First see `uml.ml`. `UML` stands for `U`ntyped `ML`, and you will be implementing an interpreter of UML which is another name of the (untyped) λ-calculus.

```
type var = string
type exp =
    Var of var
  | Lam of var * exp
  | App of exp * exp
```

The datatype `exp` corresponds to the syntactic category expression in the course notes:

- `Var` x denotes a variable x as an expression in UML.

- `Lam (`x`, `e`)` denotes a λ-abstraction $\lambda x.\, e$ in UML.
- `App (`e_1`, `e_2`)` denotes an application $e_1\ e_2$ in UML.

Next see `eval.mli` and `eval.ml`. The goal of this assignment is to implement function `stepv`:

```
(* one-step reduction in the call-by-value reduction strategy,
   raises Stuck if impossible *)
val stepv : Uml.exp -> Uml.exp
```

That is, `stepv` takes an expression e of type `Uml.exp` and return another expression e' that e reduces to; if there is no such expression e', an exception `Stuck` is raised. `stepv` uses the call-by-value strategy.

After implementing the function `stepv` in `eval.ml`, run the command `make` to compile the sources files.

```
$ make
ocamlc -thread  -c uml.ml -o uml.cmo
ocamlyacc parser.mly
3 shift/reduce conflicts.
ocamlc -c parser.mli
ocamlc -c parser.ml
ocamllex lexer.mll
15 states, 929 transitions, table size 3806 bytes
ocamlc -c lexer.ml
ocamlc -thread  -c inout.ml -o inout.cmo
ocamlc -thread  -c eval.mli -o eval.cmi
ocamlc -thread  -c eval.ml -o eval.cmo
ocamlc -thread  -c loop.ml -o loop.cmo
ocamlc -thread  -c hw4.ml -o hw4.cmo
ocamlc -o lib.cma -a uml.cmo parser.cmo lexer.cmo inout.cmo eval.cmo loop.cmo
ocamlc -o hw4 lib.cma hw4.cmo
```

There are two ways to test your code in `eval.ml`. First you can run `hw4`. At the UML prompt, enter a UML expression followed by the semicolon symbol `;`. (The syntax of UML will be given shortly.) Each time you press the return key, a reduced expression according to your `stepv` function is displayed.

```
$ ./hw4
Uml> (lam x. x) (lam y. y);
((lam x. x) (lam y. y))
Press return:
(lam y. y)
Press return:
```

Alternatively you can use those functions in `loop.ml` in the interactive mode of OCAML. (You don't actually need to read `loop.ml`.) At the OCAML prompt, type `#load "lib.cma";;` to load the library for this assignment. Then open the structure `Loop`:

```
$ ocaml
        OCaml version 4.05.0
```

```
# #load "lib.cma";;
# open Loop;;
#
```

You type `loop (step Eval.stepv (wait show));;` at the OCAML prompt, and then enter a UML expression followed by the semicolon symbol `;`.

```
# loop (step Eval.stepv (wait show));;
Uml> (lam x. x) (lam y. y);
((lam x. x) (lam y. y))
Press return:
(lam y. y)
Press return:
Uml>
```

Each time you press the return key, a reduced expression is displayed. You can try `loop (step Eval.stepv show);;` to skip all intermediate steps. Or you may use a UML expression stored in a separate file. We provide three UML files: `nat.uml`, `rec.uml`, and `fib.uml`.

```
# loopFile "nat.uml" (step Eval.stepv (wait show));;
...
```

If you want to see the entire reduction sequence without pressing the return key, use `step Eval.stepv show`:

```
# loopFile "nat.uml" (step Eval.stepv show);;
...
```

If you want to skip all intermediate steps and see only the final result, use `eval Eval.stepv show`:

```
# loopFile "nat.uml" (eval Eval.stepv show);;
...
```

The syntax of UML

The syntax for `UML` closely resembles that for the λ-calculus. The only difference is the use of the keyword `lam` in place of λ, and syntactic sugar `let` x `=` e `in` e' for `(lam` x`.` e'`)` e.

$$\text{expression} \qquad e \quad ::= \quad x \mid \texttt{lam}\ x\texttt{.}\ e \mid e\ e \mid \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e$$

The following UML expression computes a Church numeral for a natural number eight (which is found in `nat.uml`):

```
let one = lam s. lam z. s z in
let add = lam x. lam y. lam s. lam z. y s (x s z) in
let two = add one one in
let four = add two two in
let eight = add four four in
eight
;
```

Submission instruction

1. Make sure that you can compile `eval.ml` by running `make`.
2. When you have your file `eval.ml` ready for submission, copy it to your hand-in directory on `programming2.postech.ac.kr`. For example, if your Hemos ID is `foo`, copy it to:

   ```
   /home/class/cs321/handin/foo/
   ```